BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2020 Bc. Roman Ligocki

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS
ÚSTAV TELEKOMUNIKACÍ

SECURE COMMUNICATION WITHIN THE PX4 PLATFORM
ZABEZPEČENÁ KOMUNIKACE V RÁMCI PLATFORMY PX4

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Roman Ligocki**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Petr Číka, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study field **Information Security**

Department of Telecommunications

Student: Bc. Roman Ligocki ***ID:*** 169280

Year of study: 2 ***Academic year:*** 2019/20

TITLE OF THESIS:

Secure communication within the PX4 platform

INSTRUCTION:

Study the issue of drone control with a narrow focus on the PX4 platform and the Mavlink protocol. Perform a complete analysis of communication security together with the vectorization of attacks. Design security for individual parts of the communication. The design will be implemented and verified via test scenarios. The result will be a solution that provides secure transmission between the control unit and the drone. Emphasis will be placed on efficiency, simplicity and low energy consumption.

RECOMMENDED LITERATURE:

[1] PETROVSKY, Oleg. Attack on the drones. In: Virus Bulletin Conference. 2015.

[2] ALLOUCH, Azza, et al. MAVSec: Securing the MAVLink Protocol for Ardupilot/PX4 Unmanned Aerial Systems. arXiv preprint arXiv:1905.00265, 2019.

Date of project specification: 3.2.2020 ***Deadline for submission:*** 1.6.2020

Supervisor: doc. Ing. Petr Číka, Ph.D.
Consultant: Ing. Radek Fujdiak, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
Subject Council chairman

ABSTRACT

PX4 platform is one of the most common software packages used to control unmanned systems. It uses the MAVLink protocol for communication between the autopilot, ground control station or other devices in the MAVLink network. This protocol is specially designed to suit unmanned systems using radio with low passthrough. With the rising number of unmanned systems, the number of cyber attacks on these devices is also increasing. This diploma thesis presents an analysis of the MAVLink protocol and PX4 platform, and describes possible security vulnerabilities in telemetry connection. Based on these findings, a security implementation was created. This implementation contains encryption, access control, authentication and a key exchange system. Security implementation is based on the MonoCypher cryptography library. The whole implementation was programmed in C language. Author's goal was to share results with the community around PX4 platform. Therefore, pull requests have been created to the public repository during the final part of thesis.

KEYWORDS

PX4, MAVLink, unmanned systems, encryption, authentication, access control, key exchange, security, implementation,

ABSTRAKT

PX4 platforma je jedna z nepoužívanějších softwarových balíčků pro řízení bezpilotního systému. Používá MAVLink protokol pro komunikaci mezi autopilotem, pozemní stanicí a dalšími zařízeními v MAVLink síti. Je speciálně navržen pro bezpilotní systémy používající rádia s nízkou datovou propustností. S rostoucím počtem těchto zařízení dochází rovněž k růstu počtu útoků na tyto systémy. Tato diplomová práce obsahuje analízu a popis bezpečnostních nedostatků v telemetrické komunikaci platformy PX4 běžící na protokolu MAVLink. Na základě těchto nedostatků byla dále navržená a implementována bezpečnostní řešení. Tato implementace zahrnuje šifrování, řízení přístupu, autentizaci a systém pro výměnu klíčů. Bezpečnostní implementace je postavená na knihovně MonoCypher. Všechny části práce jsou naprogramováno v jazyce C. Cílem autora je sdílet výsledky, kterých dosáhl s komunitou kolem paltformy PX4. Proto během finální části práce vznikl pull request do veřejného repozitáře.

KLÍČOVÁ SLOVA

PX4, MAVLink, bezpilotní systém, šifrování, authentizace, řízení přístupu, výměna klíčů, bezpečnost, implementace

LIGOCKI, Roman. *Secure communication in PX4 platform*. Brno, 2020, 66 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications. Advised by doc. Ing. Petr Čika, Ph.D.

ROZŠÍŘENÝ ABSTRAKT (CZECH EXTENDED ABSTRACT)

Úvod:

Platforma PX4 je jedna z nejpouživanějších softwarových balíčků pro řízení bezpilotních systémů. Zahrnuje firmware pro desku plošných spojů, komunikační protokol MAVLink a software QGroundControl pro plánování autonomních misí a konfiguraci bezpilotního systému. Platforma PX4 umožňuje uživateli řízení velkého množství typů bezpilotních systémů. Mezi tyto systémy můžeme zařadit například tradiční letadlo, samokřídlo, multi-koptéry, letadlo s kolmým vzletem, auto, pásové vozidlo nebo ponorky. S rožšířujícím se trhem součástek umožňujícím konstrukci bezpilotního systému s pouhou základní znalostí elektrotechniky dochází k nárustu počtu těchto bezpilotních systémů. S jejich rostoucím množstvím rovněž dochází ke zvýšení počtů útoků na ně. Je proto potřeba zajistit, aby zařízení nebylo možné bez oprávnění ovládnout a způsobit majetkové nebo duševní škody.

Řešení:

Platforma PX4 pro komunikaci mezi autopilotem, pozemní stanicí a dalšími zařízeními používá protokol MAVLink. Jedná se o komunikační protokol navržen pro potřeby řízení bezpilotních systémů používající telemetrické rádio s nízkou propustností. Mezi hlavní výhody protokolu MAVLink patří malá velikost hlavičky paketu. Pro vytvoření prázdného paketu je potřeba pouze jedenáct bytů. MAVLink protokol a platforma PX4 byly v této diplomové práci blíže zanalyzovány z pohledu principu fungování autopilota, různých typů telemetrických rádií a komunikace mezi řídicí stanicí a autopilotem. Následně na základě lepší znalosti celé platformy došlo k nalezení možných vektorů útoků na telemetrické propojení mezi autopilotem a pozemní stanicí. Dále z těchto zjištění byl odvozen návrh bezpečnostní implementace. Tento návrh zahrnoval šifrování, řízení přístupu, autentizaci a systém výměny klíčů. Pro implementaci všech zmíněných kryptografických prvků bylo potřeba nalézt vhodnou kryptografickou knihovnu. Parametry při výběru této knihovny byly programovací jazyk C, jednoduchost použití, kompatibilita mezi různými architekturami procesorů a nízká výpočetní náročnost. Do užšího testování byly vybrány knihovny MonoCypher a LibHydrogen. Po následné testovací implementaci, kde došlo k praktickému ověření rychlosti jednotlivých algoritmů a vlastností knihoven, byla vybrána knihovna MonoCypher. Ve všech testech rychlostí kryptografických algoritmů měla knihovna MonoCypher lepší výsledky než knihovna LibHydrogen. Další důvod proč byla vybrána knihovna MonoCypher byl, že knihovna LibHydrogen ukládá validační MAC část ve stejné proměnné jako šifrový text. Tato vlastnost LibHydrogen znemožňovala realizaci navržené implementace na protokolu MAVLink. Další důležitou částí práce bylo ověření zdroje entropie na různých zařízeních plat-

formy PX4. Mezi otestovaná zařízení patří Pixhawk 1 a CubePilot Black. Rovněž byl otestován zdroj entropie na počítači MacBook Pro 2015 s procesorem Intel. Pro ověření kvality zdroje entropie byla použita sada testů od institutu NIST. Tato sada obsahuje patnáct různých testů, kterých cílem je zjistit, zda generování za sebou jdoucích bitů je náhodné.

Výsledky:

V rámci praktické části práce byla do repozitáře MAVLink přidána nová zpráva CERTIFICATE. Tato zpráva zahrnuje veřejný klíč pro systém výměny symetrického klíče, podpis certifikační autority a informace o úrovni oprávnění, vlastnictví a názvu bezpilotního systému. Tato zpráva byla navržená tak, aby ji bylo možné v otevřené podobě sdílet s jinými systémy. Po vzájemném přijetí CERTIFICATE zpráv je možné ověřit, zda druhá strana má certifikát podepsaný validním soukromým klíčem, a následně pomocí algoritmu X25519 vygenerovat společný symetrický klíč. Celá implementace byla následně naprogramována v jazyce C. Největší počet změn v zdrojovém kódu byl proveden v repozitáři ArduPilot/pymavlink. V tomto repozitáři se nachází kód zajišťující parsování, odesílání a přijímání zpráv MAVLink. Těmito změnami došlo k rozšíření patičky MAVLink paketu o dalších 40 bytů v každé zprávě. To způsobilo nárůst objemu běžné komunikace o přibližně padesát procent. Výsledkem praktické části práce je přidání end-to-end šifrování, ověřování zpráv a výměny klíčů do platformy PX4. Cílem autora bylo hotovou práci sdílet s komunitou tvořící platformu PX4. Z toho důvodu došlo k závěru práce k vytvoření pull requestů do veřejného repozitáře. To bude vyžadovat možné přizpůsobení a další úpravy implementace dle případných připomínek.

Závěr:

V rámci práce došlo k nalezení bezpečnostních nedostatků komunikačního protokolu MAVLink, který se používá mezi softwarem pozemní stanice QGroundControl a firmwarem autopilota PX4. Následně došlo k nalezení řešení těchto nedostatků pomocí návrhu bezpečnostní implementace. V závěru práce je popsán postup výsledné implementace a ověření její funkčnosti. Tímto došlo ke splnění zadání této diplomové práce.

DECLARATION

I declare that I have written the Master's Thesis titled "Secure communication in PX4 platform" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

I would like to pay my special regards to Ing. Radek Fujdiak Ph.D. for consultations and all help during the writing of my thesis. Also I wish to show my gratitude to PX4 developers Julian Oes and Hamish Willee, for valuable advice without which the implementation would not be created. Lastly I would like to recognize the invaluable assistance of my family that they provided during my studies.

Contents

List of Figures

List of Tables

Introduction

Nowadays, it is easy to build custom designed unmanned aerial vehicles. Different types of components used to build UAVs, like autopilots, *Radio Controlled* (RC) transmitters, ESCs, motors, batteries, or propellers, are available for a reasonable price. There is also a massive amount of open-source software platforms used to control these autopilots. It only take little time to learn the basics of RC systems to get enough information to find out which hardware you need and how to connect these components. It means that nearly anyone can have their own unmanned vehicle to fly for fun, to record video with or take aerial images. Using widely available hardware, users can create different types of vehicles like planes, multi-rotors, *Vertical Takeoff and Land* (VTOL) planes, and tracked or wheeled cars. This means that having drones is no longer the privilege of research institutions or companies. The drone market already expects that the number of drones will be rising [1].

An increasing number of flying unmanned aerial vehicles might lead to a rise in the interest of hackers in this technology. This could lead to the discovery of ways of how to unlawfully take control of these vehicles away from the owner. The hacked vehicles could then become potentially dangerous to people and structures in the vicinity. It is also possible for these vehicles to have an expensive or important payload that might be stolen or even destroyed. An example of such payload could be blood packages or organs for donation. The Zipline company delivers blood packages in Rwanda, where blood distribution using aerial vehicles is more secure, reliable, and faster than transporting using cars [2]. To make these delivery systems less prone to be hacked, unmanned vehicles should use communication protocols with strong confidentiality and integrity of internal messaging, or at least use a secure communication channel.

The PX4 platform uses the MAVLink communication protocol to exchange information like altitude, direction of flight, rotation in all axes or mission plans. The first goal of this thesis is to examine the PX4 platform and discover possible vulnerabilities in the PX4 stack and the MAVLink protocol. The second goal is to find solutions to these issues and fix them using a newly created security implementation. This new implementation should eliminate all possible ways of seizing control of the vehicle by an unauthorized person, stealing of confidential information, or making the vehicle dangerous to its surroundings. The final security design should be integrated into the PX4 software stack and the MAVLink protocol via pull requests into public repositories in order to share the results of this thesis with the community around the PX4 platform, and to make unmanned vehicles more secure all around the world.

1 PX4 Software stack

PX4 is an open-source community based autopilot platform, founded by Lorenz Meier in 2008. At first known as the "ETH Pixhawk project", it was renamed to PX4 when it was made available to the public in 2011 during Lorenz Meier's doctorate studies [3]. The goal of his team was to create *Unmanned Aerial Vehicle* (UAV) with autonomous flight features based on computer vision.

Today, PX4 is an industry-leading platform that allows companies to create their UAVs for commercial usage. Components that work with PX4 are also affordable for hobbyists. This allows amateurs to easily construct UAV with cheap hardware that can be powered by an open-source software stack like the PX4. Nearly anyone with basic knowledge of computer science and electronics can build a quad-copter that can carry some amount of payload. All necessary information about the construction and configuration of new drones are available in the PX4 online user guide [4]. This documentation provides a large amount of tutorials describing how to configure each part of an UAV.

According to the user guide [4], one of the biggest advantages of PX4 is that it allows developers and users to create not only traditional multi-copters, planes, rovers, VTOL planes, but also provides the option for a user to define custom frame designs using only a few lines of code in the actuator mixer. All these vehicles can be fitted with different types of payloads, e.g. cameras for video or image capture, lidars, or multispectral cameras. This might create opportunities for new businesses to use these drones for precise agriculture, geodesy, during search and rescue operations, or for military usage.

PX4 is not the only open-source platform for unmanned systems. There is also an ArduPilot project that was created in 2009. Today, ArduPilot, together with PX4, is one of the most used platforms for a variety of different vehicles. ArduPilot also has good user documentation [5]. According to both documentations [4][5], PX4 and ArduPilot use the same MAVLink protocol to provide telemetry connection between UAV and Mission planer software. Because of that, it is possible to communicate with both autopilot platforms using the QGroundControl software.

At the beginning of this thesis, it is important to point out one thing. Most people use the word "drone" incorrectly. All flying machines without pilots are called UAVs. Only UAVs that can fly autonomously can be called drones [6]. In this thesis, the word "drone" will not be used, because using PX4, you can build UAVs or *Unmanned Ground Vehicle* (UGV) without autonomous control.

The entire PX4 Flight stack, which includes autopilot firmware, is licensed under the BSD 3-clause license [7]. This license is one of the least constrained licenses that can be used, as it allows users and programmers to copy code with only a single

limitation – the developer needs to insert the original author and project name into one's source code. There is no limitation on the number of software copies, nor on possible future development, which includes modifications of the original source code. As a user or developer, you do not need to share the modified source code like with the GNU *General Public License* (GPL) license. This means that this license is perfect for proprietary forks of open-source software. This permissive license allows companies, for instance, to use this open-source code without the need to share their implementations of some secret features.

1.1 PX4 autopilot architecture

PX4 autopilot firmware is one of the most important parts of the whole PX4 platform. This software is something like a pilot that watches all sensors' values and listens to commands. Based on that data, it generates outputs to all actuators to control the unmanned vehicle. Autopilot is based on two layers: Flight stack and Middleware.

Flight stack is doing sensors and user input fusion, estimations, and flight control. Middleware creates a layer that handles all internal and external communication, but also provides drivers for supported hardware. The flight stack layer is a collection of all estimation and control algorithms. There are controller algorithms for all the different kinds of vehicles like rovers, multi-rotors, helicopters, and VTOL planes. With these algorithms, it is possible to maintain the position and attitude of the vehicle during flight. Flight stack diagram is shown on Fig. 1.1. During this thesis, version of PX4 autopilot firmware was 1.10.0.

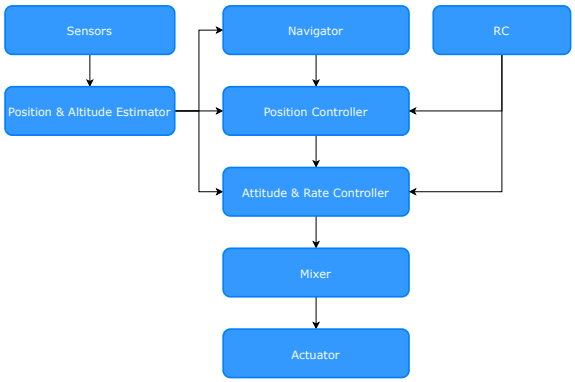


Fig. 1.1: PX4 flight stack diagram [8]

The middleware layer includes drivers necessary to maintain communication and drivers for all supported sensors. It also provides communication tools for all peripherals such as *Global Positioning System* (GPS) boards, external magnetome-

ter, or companion computer. To make the communication between different parts of autopilot firmware simple for developers, middleware has has integrated uORB message bus that works on the public-subscription system. This provides a solution to share data between services running on PX4 autopilot firmware. For instance, uORB is used when the developer needs to get sensor information into the service, maintaining the attitude of UAV. For example, in PX4, there is a subscription topic sensor_mag where raw data from magnetometer are published. Any service that needs these data can pull them using a subscription system. It also gets a notification when published data are updated.

1.2 QGroundControl software

To set up and control the unmanned system based on PX4 platform or ArduPilot, the user needs to have some software that can communicate with that UAV. These types of software are called *Ground Control Station* (GCS) or mission planners. In the case of PX4 and ArduPilot, there is a software called QGroundControl. It is a multi-platform mission planner that allows users to set up and control their vehicles, change configurations, analyze flights, and also provide a way to plan autonomous missions. Also, it is possible to view telemetry data and the current position or status of the plane. All information about QGroundControl is available on the user guide website [9]. The final implementation was tested on QGroundControl version 3.5.6.

The main advantage of QGroundControl in comparison to other mission planners is that it runs on nearly every modern operating system like Windows, Linux, macOS, Android, but also on iOS and iPadOS. This is possible because QGroundControl is based on the Qt framework (Qt for application development). With Qt *Application Programming Interface* (API) included in the Qt framework, it makes the final code runnable on nearly any device.

Software that is build using the Qt framework has only two options in case of license. Qt provides commercial and open-source licenses. For example, if a company wants to develop proprietary software, it needs to pay for every software developer that works on that software. If the company doesn't want to pay for the Qt developer license, then it needs to provide source code of their software because final software must be licensed under GPL or *Lesser General Public License version 3* (LGPLv3).

2 PX4 Hardware

Another part of the PX4 platform is the Flight controller. It is a *Printed Circuit Board* (PCB) with a micro-controller, all the necessary sensors and connectors to fly and control UAV. Most commonly used Pixhawk 1 is based on *Flight Management Unit version 2* (FMUv2) open hardware design. FMU provides a standard for supported sensors, micro-controllers, outputs, communication buses, and memory and flash storage size.

Higher versions of FMU have newer sensors, more storage for flight firmware, more computation performance, and features like sensors redundancy. All these improvements add new possibilities for additional features. In version 1.10, newer FMUs comes with better refresh rates of multi-copter attitude stabilization. There are also different FMUs that are supported by PX4, but their hardware designs are not open-source. Information about all supported hardware is available in the developer guide [10].

There is lot of manufacturers of PX4 flight controllers:

- 3DR (Pixhawk 1),
- mRobotics,
- HobbyKing,
- Holybro,
- Drotek,
- Hex (CubePilot Black).

To control actuators like servos or ESCs, every PX4 autopilot board has some output ports. On the Pixhawk board series, there are two ways to output actuators: using a digital signal or over a communication bus. The most used digital signal is *Pulse Width Modulation* (PWM), that might be based on the duty cycle in a defined period or One-Shot or D-Shot protocols. One-Shot or D-Shot protocol based on PWM signal offer higher signal precision and additional robustness. It is also possible to use *Universal Asynchronous Receiver-Transmitter* (UART) or *Controller Area Network* (CAN) buses to send output to actuator. Commonly used is PWM signal based on the duty cycle, which is supported by dozens of servos or ESCs. All implementations and tests during this thesis will be tested on CubePilot Black and Pixhawk 1 boards, which will be described in more detail in the next section.

2.1 Pixhawk 1

Pixhawk boards are one of the commonly used autopilot boards design for PX4 or ArduPilot platforms. With the price of about 129 USD for original Pixhawk 1 manufactured in the United States of America or one of the many clones that cost

about 50 USD, it is one of the most used autopilot hardware on the market. It has all the necessary sensors that UAV needs to control the attitude and altitude of the vehicle. This board does not have any radio hardware or GPS receiver included onboard. To connect peripherals like GPS receiver, RC links or telemetry link, there are DF13 connectors, that are widely used in UAV industry.

This board is targeted mainly for hobby usage. The disadvantage of this board is that it does not include features like *Inertial Measurement Unit* (IMU) temperature stabilization or IMU redundancy. All provided information are based on the 3DR Pixhawk hardware manual [11].

Hardware specification:

- 32-bit STM32F427 Cortex M4 core with FPU,
- 168 MHz/256 KB RAM/2 MB Flash,
- 32-bit STM32F103 failsafe co-processor.

IMU specification:

- 3-axis 16-bit gyroscope L3GD20,
- 3-axis 14-bit accelerometer/ magnetometer LSM303D,
- Invensense MPU 6000 3-axis accelerometer/gyroscope,
- MEAS MS5611 barometer.

IO specification:

- 1x I2C (separate connectors),
- 2x CAN: CAN1 and CAN2,
- 5x UART: TELEM1, TELEM2, GPS, SERIAL4, SERIAL5,
- 1x HMI: USB extender,
- 14xPWM output,
- 1x RC input,
- 1x RSSI input.

Fig. 2.1: Pixhawk 1 autopilot board [11]

2.2 CubePilot Black

CubePilot Black is a more advanced autopilot, that is widely used in industrial and commercial systems. Name "Cube" is based on a two-part design, where one part is a carrier board that provides all the wiring for the autopilot, and the second part is the CPU board with IMUs installed in a small cubic shape enclosure with integrated heater and vibration isolation. The cubic enclosure with the CPU and IMUs is removable. With features like triple-redundant IMU, vibration stabilization, and temperature stabilization, this board is perfect for commercial and industrial usage. This board, based on FMUv3, also provides more memory and Flash storage. This means that developers can create more complex control and estimation algorithms. The board comes with the price of 238 USD per set. All provided information are based on the Hex CubePilot Black hardware manual [12] and [13].

Hardware specification:

- 32bit STM32F427 Cortex-M4F® core with FPU,
- 168 MHz / 252 MIPS,
- 256 KB RAM,
- 2 MB Flash (fully accessible),
- 32 bit STM32F103 failsafe co-processor.

IMU specification:

- Onboard fixed IMU,
  - 3-axis gyroscope / accelerometer MPU9250,
  - barometer MS5611,
- Two vibration isolated and heated IMU.
  - 3-axis accelerometer/magnetometer LSM303D,
  - 3-axis gyroscope L3GD20,
  - 3-axis gyroscope / accelerometer MPU9250 or ICM 20xxx,
  - barometer MS5611.

IO specification:

- 2x I2C,
- 2x CAN: CAN1 and CAN2,
- 5x UART: TELEM1, TELEM2, GPS (I2C 1 embedded), SERIAL4(I2C 2 embedded), SERIAL5,
- 1x HMI: USB extender,
- 14x PWM output,
- 1x RC input,
- 1x RSSI input.

Fig. 2.2: CubePilot Black autopilot board [12]

2.3 Hardware setups

This section presents four examples of hardware setups that are used by hobbyists and professionals. All of them contain autopilot that is necessary to run PX4, telemetry radio set for MAVLink protocol communication, and RC radio to control the rover. RC radio is commonly secured using a proprietary communication system created by the manufacturer on transmitter and receiver. Because of this, more details will be provided only on the case of MAVLink connection systems and their security issues. Examples of hardware setups might be seen in Tab. 2.1.

2.4 PX4 compatible radios

In this section, the radios mentioned earlier will be introduced. They are working on different frequencies. If a device is considered "legal", then it means that this device is compatible with regulations in the Czech Republic. Most of them are configurable, so based on different regulations, it may be possible to use them also in other countries. Other countries were not part of the research. Regulations to use free frequencies in the Czech Republic are described in the document released by *Czech Telecommunication Office* (CTO) [14].

2.4.1 SiK telemetry radio

SiK telemetry radios are devices based on open-source SiK firmware and cheap SiLabs S1000 *System on a Chip* (SoC) [15]. As an example, Holybro telemetry radio V3 433 MHz set consists of two devices. Both of them have micro-USB connector for PC or tablet connection and 6pin JST-GH connector to connect the radio to Pixhawk 2.4.8 or another autopilot. With the included antenna and with a default configuration, it is possible to control the drone at a 300 m distance. With default configured output power, that is 100 mW, so it is not legal to use this device in the

Tab. 2.1: Example of hardware setups, *RFD TXMOD passthrough PPM signal from Frsky RC transmitter to RFD 868x air receiver. There is no need to have RC receiver. **It is possible to configure RFD 868x and RFD TXMOD to encrypt communication using AES algorithm with pre-shared key. ***No information provided by manufacturer.

Parts		Hobby #1	Hobby #2	Profi #1	Profi #2
Autopilots	Pixhawk 1	Yes	Yes		
	CubePilot Black			Yes	Yes
RC receivers	FlySky FS-iA6	Yes			
	Frsky X8R		Yes		
RC transmitter	FlySky FS-i6	Yes			
	Frsky Taranis X9D		Yes		
	Frsky Horus X10S			Yes	Yes
Telem. radio, ground	PC + Sik radio	Yes			
	PC		Yes		
	iPad+RFD TXMOD			Yes*	
Telem. radio, air	Sik radio	Yes			
	Wifi radio		Yes		
	RFD 868x			Yes	
All-in-One	HereLink				Yes
Vulnerabilities	Encryption	No	No	Optional**	/***
	Authentication	No	No	No	No
	Access control	No	No	No	No

Czech Republic. If this radio were configured to transmit an output signal with a power of 10 mW, it would be legal. Fig 2.3 shows SiK radio setup.

In the case of secure communication, SiK telemetry radios made by Holybro do not provide any software configuration enabling encryption and decryption of UART data channel. Using a another set of those radios, it is possible to receive data from other transmitters.

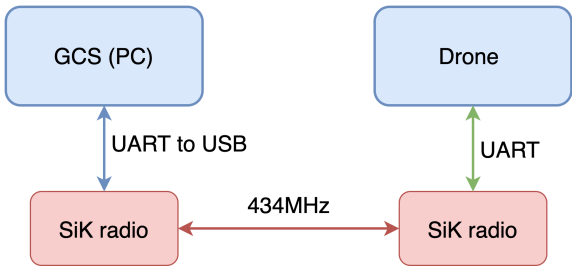


Fig. 2.3: Diagram how SiK radio is connected

2.4.2 Wi-Fi telemetry radio

This telemetry device is using *Wireless Local Area Network* (WLAN) to create a network where it broadcasts MAVLink packets encapsulated into TCP/IP. Setup is effortless, because on the ground station side there is no need to have any other device except the computer or tablet with Wi-Fi. Wi-Fi telemetry radio works as a router. It has a running DHCP server, so there is no additional configuration needed. All messages and commands coming from UART are encapsulated into TCP/IP packet and broadcasted to active connected devices on whole network. Fig. 2.4 shows connection of Wi-Fi telemetry radio.

This solution is compatible with operating systems like Mac, Linux, or Windows. Also, it is possible to connect to this telemetry radio using an Android or iOS device with QGroundControl. With output power of 100 mW, it is legal to use this telemetry device in the Czech Republic. If *Wi-Fi Protected Access 2* (WPA2) is enabled, all telemetry data are encrypted using the pre-shared key with encryption algorithm AES-CCMP. Communication is secure unless the pre-shared key for WPA2 is revealed. Then any MAVLink packet could be captured from UDP communication, decrypted, and decoded.

2.4.3 RFD868 combo

RFD868 is one of the best solutions to communicate with unmanned systems [16]. It provides a very reliable and long-range telemetry link. It is possible to create a telemetry link up to 80 km using pitch antennas. These radios also have the ability

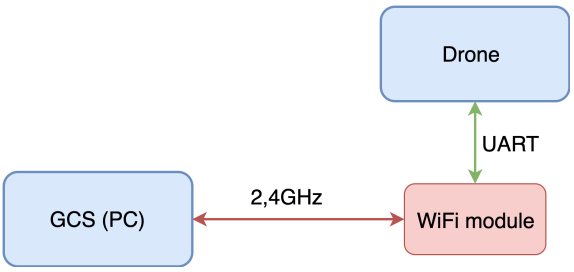


Fig. 2.4: Diagram showing how the Wi-Fi telemetry radio is connected

to passthrough *Pulse Phase Modulation* (PPM) signal to control the drone. All RFD868 radio sets allow us to encrypt all MAVLink and PPM data using *Advanced Encryption Standard* (AES) encryption algorithm. Diagram of RFD868 setup is possible to view in Fig. 2.5

There is also a modified TXMOD package that contains RFD868 and might be connected to the JR socket in Frsky's RC radios. This TXMOD package also contains a Wi-Fi access point that creates a Wi-Fi network around TXMOD. This way, any Wi-Fi compatible device is able to connect to this network and then receive MAVLink communication. In the Czech Republic, it is legal to use this device with output power up to 500 mW, but only on one channel (869.4 - 869.65 MHz) and with only 1% of the duty cycle. This configuration will allow us to communicate over very long ranges but with low throughput.

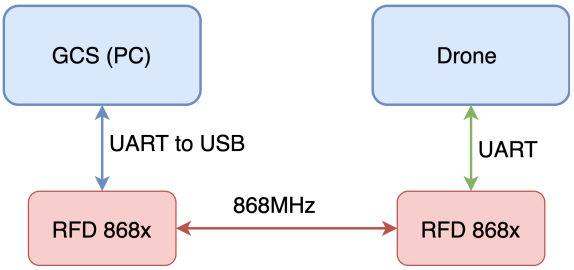


Fig. 2.5: Diagram how RFD868x telemetry radio is connected

2.4.4 Herelink HD Video transmission system

Herelink is an all-in-one solution that provides a long-range link (up to 16 km) for telemetry, PPM signal, but also provides a video link for all connected HDMI cameras. Herelink ground station is an android tablet with a touch screen that has two 2-axis joysticks, wheel, and six buttons to control QGroundControl software. The radio link is integrated, and there is no need to buy additional hardware to control the unmanned system. The Air unit contains two HDMI input ports, where the user can insert the output from any HDMI compatible camera. Video is transmitted with

compression based on signal quality to the ground station. Fig. 2.6 shows diagram of Herelink setup. For *First Person View* (FPV) video output, you need to use the Analog/HDMI converter. Currently, QGroundControl is only GCS software, that is available for Herelink. There is no option to customize QGroundControl because this modified version for Herelink is not open-source. More information might be found in Herelink manual [17].

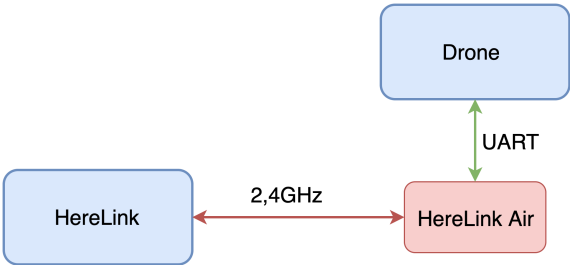


Fig. 2.6: Diagram how Herelink radio is connected

2.4.5 RFD868 in Multi-point mode

As it is described in manual [16], RFD868 modem can create a very reliable and fast connection between two modems. It has also feature to create a multi-point network or asynchronous non-hopping mesh. The difference between these two modes is that multi-point networks does not allow to communicate with a node that is not in range of the base node. In asynchronous non-hopping mesh mode, it is possible to communicate with multiple nodes outside of local node range. If unreachable node is within reach of other node, data might be retransmitted.

2.4.6 Summary

As described in this section, there are radio sets that allow encrypting the communication channel on the transport layer. This provides confidentiality of the data transmitted between multiple trusted devices. All of these radios lack an authentication system that would provide a solution to authenticate messages.

Because of this, it was decided to create an implementation of encryption, authentication, and access control systems on the application layer in the MAVLink protocol. In the next chapter, the MAVLink protocol will be introduced to show details about MAVLink, which will lead to identification and fixing security vulnerabilities.

3 MAVLink protocol

MAVLink (Micro Air Vehicle link) is a lightweight communication protocol used to communicate with unmanned systems or with other devices on board. For instance, using MAVLink protocol, you can upload flight missions, get all flight data, or change parameters of autopilot on an unmanned system. MAVLink is based on a modern hybrid publish-subscribe or point-to-point design pattern. All data streams are sent/published as the topic, while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission.

As a source of information about MAVLink protocol, a MAVLink website has been used [18]. This protocol is labeled as lightweight because there is a very small overhead per packet. MAVLink 1.0 has only 8 bytes of a header for every message, whereas payload might be up to 255 bytes long. In MAVLink 2.0, there are 14 bytes of overhead, but this version provides more security and is more extensible.

All messages are defined using *Extensible Markup Language* (XML) in multiple dialects, where every message has its definition. Dialects allow creating different sets of messages for different types of systems that use the MAVLink protocol as a communication tool. Mostly used dialect is "common.xml". Example of the message written in XML format is possible to see in the appendix A. Since MAVLink protocol messages are defined in XML, it is possible to generate MAVLink API in different programming languages like C, C++, Python, Swift, C#, and others. This way, message definition is consists across multiple programming languages.

3.1 MAVLink 1.0

MAVLink 1.0 was first released in 2009 by Lorenz Meier. His goal was to create a very reliable communication protocol for varied types of vehicles, communication environments (radios with low throughput or high latency/noise channels). It also provides a detection system for lost or corrupted packets.

MAVLink 1.0 needs a very small amount of management. Only 8 bytes are needed to create a packet with no payload. Nowadays, MAVLink is ported for many different programming and interpreting languages like C, C++, Python, Java, and Swift, and is also capable of running on many computer architectures and operating systems (ARMv7, ATMega, dsPic, STM32, Windows, macOS, Linux).

Header description:

- STX – Start byte of value 0xFD,
- LEN – Number of bytes saved in PAYLOAD part of a packet,
- SEQ - Sequence number of the packet. Provides a way to detect packet loss,
- SYS ID – System ID of sending device. Used to address device in a network,

- COMP ID – Component ID of sending device. Indicates the type of device in network,
- MSG ID – Message ID. Needed for payload serialization,
- PAYLOAD – Message serialized into an array of bytes,
- CHECKSUM – Cyclic redundancy check. Used for corruption detection.

3.2 MAVLink 2.0

Compared to the older version of MAVLink, version 2.0 provides new features like compatibility, incompatibility flags, and extended message ID. These features allow developers to create more types of messages and also provide flags that indicate if the packet should be handled in a different way than a regular packet. Also, MAVLink 2.0 adds packet signing, that provides basic authentication.

An example of an incompatibility flag might be the signature MAVLINK_IFLAG_SIGNED flag, which adds information that during packet receiving additional 13 bytes needs to be received. Those 13 bytes will be used for signature transmission.

The next feature is empty-bytes payload truncation, which removes the last zero bytes from the serialized payload. Using this technique, MAVLink 2.0 lower the number of payload bytes sent over the communication channel. MAVLink 2.0 is still not fully ported into all programming languages supported by MAVLink 1.0.

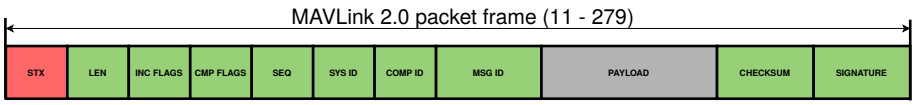


Fig. 3.1: MAVLink 2.0 packet frame [18]

Compared to MAVLink 1.0 in MAVLink 2.0, there were added new bytes into the header that allow new features.

- STX byte was changed from 0xFD to 0xFE.
- INC (Incompatibility) FLAGS byte was added.
- CMP (Compatibility) FLAGS byte was added.
- MSG ID was enlarged from one byte into three bytes.
- SIGNATURE array was added.

3.2.1 Packet signature

MAVLink 2.0 adds a message signing feature, that allows authenticating if the message comes from a reliable source. To do this, SIGNATURE part in MAVLink 2.0

was added, where a sign is stored. This creates security layers where fake attacker's messages are not accepted on the receiver side because there is no way to create a valid signature for a fake message without the pre-shared secret key.

To enable signature in MAVLink 2.0 packet, it is necessary to enable an incompatibility flag MAVLINK_IFLAG_SIGNED to inform the receiver that an additional 13 bytes need to be received. After inspection of source code in current version of MAVLink 2.0 an security vulnerability was found, that does not allow user the option to unconditionally reject messages that are unsigned. This allows attackers to just send an unsigned message. Without validation of the condition that only signed messages will be received, it is possible to use this vulnerability. In the case of implementation, it should be easy to fix this problem.

SIGNATURE part in MAVLink 2.0 packet has three parts:

- LinkID – ID of a link on which message is sent
- Signature – Six-byte signature computed using SHA256 algorithm
- Timestamp – Six-byte number with units of 10 microseconds since 1st January 2015 GMT

Signature works based on the *Keyed-Hashing for Message Authentication* (HMAC) principle and is computed based on the *Secure Hash Algorithm* (SHA) 256 algorithm. Commonly SHA256 hash algorithms return an array of bits with a length of 256. To lower the length of the signed packet, the signature of MAVLink is shortened to only 48 bits. In case of security, a shorter signature leads to losing signature strength. There is a bigger probability ($1/2^{48}$ bit) to find out by brute force a collision signature that will suit malicious message. The formula of how the signature is computed can be seen on equation 3.1.

$$Sig = TO48(SHA(K + H + P + CRC + ID + TS)) \tag{3.1}$$

where TO48 is a function to cut first 48 bits from a 256-bit hash, SHA is SHA256 hash function, K is symmetric key bytes, H is a header of MAVLink packet, P is payload bytes, CRC is checksum bytes of MAVLink packet, ID is system ID byte of sender device, and TS is timestamp bytes.

The signed message is accepted only under these conditions:

- Computed signature match with a signature from the received message.
- Timestamp is not older than the timestamp of the previously received packet from the same device.
- Timestamp is not older than 1 minute.

3.3 MAVLink 2.0 security issues

MAVLink protocol provides an excellent way to transport a massive amount of data with very low overhead. The problem is that it has only basic security features. In the current state of MAVLink protocol, the 2.0 version provides only a simple authentication system based on HMAC. This means that the drone maintainer must add the same symmetric key into all devices in the MAVLink network. This creates a security vulnerability, that if one of the devices is stolen, symmetric key shared between all the devices could no longer be used to have authenticated messaging. The distribution of symmetric keys might be easy to handle in the case of a few devices. If some company, for instance, uses a more considerable amount of devices, management of pre-shared symmetric keys might be very time-consuming.

The disadvantage of the current MAVLink protocol version is also lack of encryption. All data that might be sent over radio are sent as an unencrypted plain text. This allows an attacker to read all messages and to know, for instance, where exactly vehicle or drone currently is. Also, it is possible to find out what plans the pilot might have. In hobby or in commercial industry, this might not be a huge problem, but in the case of military or police usage, mission plans and information must stay confidential.

This brings the idea to add encryption into MAVLink communication and provide confidentiality to all classified data, but that brings a new challenge. Just like in the case of a signature system, symmetric encryption needs a key that must be distributed to all devices in the network. This brings the same vulnerability as in the case of a signature system, where the theft of any device from network gives access to the key to all encrypted messages. To eliminate this problem, a key exchange system needs to be implemented. In that case, a private key infrastructure might be a solution.

If there are only two devices in the MAVLink network, then encryption and signing keys distribution might be straightforward. For example, if the key is changed, there are only two devices where keys need to be changed: in ground control station and in the UAV. In the case of one hundred drone fleet, the maintainer will need to make changes on all devices. This would be very time consuming but also not a very flexible solution.

To solve this problem, there is a private key infrastructure with its key exchange system for all devices in the network. This means that every device must have its asymmetric key pair for key exchange and its certificate to identify itself and to prove that the device is part of the maintainer's fleet. The certificate might contain information about devices like name, maintainer name, public key and signature of certificate, and others.

This certificate must be signed by the authority that is trusted by all devices in the network. Using this certificate, it is possible to verify other devices using the public key of authority. This public key must be saved on all devices that want to verify other signed certificates by an authority.

The next missing part of security in MAVLink 2.0 is the access control system, which will allow the creation of groups of devices with different access rights. As an example, there might be a military group of soldiers that has a fleet of drones to operate. In that case, the MAVLink network contains five devices: Ground control station handled by the main pilot, monitor handled by the camera operator to control and view gimbal cameras. There are also three UAVs that fly different missions.

The main pilot handles mission planning using ground control station with mission planner software. In this case, there should be an access control system to authorize what each device in a network can do with other devices. It is undesirable to be able to control UAV from camera monitor, that is handled by the non-authorized person and vice versa. Also, a device like UAV should not have any rights to generate valid MAVLink message, that might change mission plans or steer other UAVs. This example shows that MAVLink needs to be able to provide different levels of access rights for any device in a network based on operator fleet architecture.

4 Cryptography libraries

To add an encryption, authentication, and access control system, there was a need to find a C library with cryptography primitives that will be easy to use and compatible with low-performance devices. There is a vast number of open-source libraries that are designed for 32-bit processors. It means that those libraries would be useful for PX4 autopilot based on Pixhawk 1.

The problem is that those libraries would not work with 64-bit Intel or AMD processors. To remove this problem, there was a need to find a multi-platform cryptography library that must be compiled on any device that could be part of PX4 setup. These types of devices have a limited performance to do all their main tasks, so if the goal is to add a security layer into communication, a cryptography library should have excellent performance. To find the best cryptography library with all necessary features, research of all publicly available with permissive license needs to be done.

In this chapter, two libraries that have been chosen by previous research will be introduced. All details of how they work will be explored to find out which of them will be the most suitable for the PX4 platform's needs.

4.1 LibHydrogen

LibHydrogen is a cryptography library inspired by the LibSodium library. It aims to be easy to use, to have good performance, and support different types of CPU architectures. Simplicity is achieved by exposing consistent high-level API instead of low-level primitives. To maintain a small footprint, it has implemented only two necessary cryptography primitives to achieve all functionality.

For key exchange, there is the Diffie-Hellman algorithm with the Curve25519 elliptic curve. For hashing and encryption, there is Gimli permutation. LibHydrogen is released under a liberal ISC license. This license is compatible with the BSD license, so in that case, if this library would be a part of PX4, there are no conflict.

One of the biggest advantages of LibHydrogen is that it has no dynamic allocation of memory during runtime. This brings more stability, which is critical in the aviation industry. The user manual is available on GitHub, where LibHydrogen is publicly available [19].

4.2 MonoCypher

MonoCypher is the next C library, that is mainly designed to be as simple as possible to use with a very small footprint. In the case of simplicity, MonoCypher is designed

the same way as LibHydrogen. Only the high-level API is available.

The size of implementation of MonoCypher library is around 2500 lines of code. Despite it's size, it provides all the necessary algorithms to have authenticated encryption, key exchange, and a signing system, but also hash functions.

In regards to the license, MonoCypher is licensed under a 2-clause BSD license. This is an even more permissive version then a 3-clause BSD license. In that case, there should be no collision with the PX4 license. The user manual for developers is available on the MonoCypher website [20].

4.3 Comparison

Both libraries are very similar, but they still have some differences. This section will provide a description of the differences between the details of the features in both libraries.

4.3.1 Features

Both libraries implement:

- Authenticated encryption,
- Hashing,
- Password key derivation,
- Key exchange,
- Public key signature,

In addition, LibHydrogen has implemented true random number generation API for many different platforms (Windows, Linux, Mac, AVR, *Advanced RISC Machine* (ARM)). The most important difference in both libraries in the case of PX4 security implementation is the key exchange system.

LibHydrogen offers three schemes to facilitate a safe exchange:

- N key exchange – In this variant, only client needs to know the public key of server. When randomly generated session key is encrypted using the public key of server, it is sent there and decrypted.
- KK key exchange –In this variant, both sides need to know each other public key. The client first sends an encrypted random value to the server. The server calculates a session key using the decrypted received random value and then sends it back to the client in the ciphertext. After the client receives and decrypts the session key, both sides have the same session key.
- XX key exchange – This variant is constructed for anonymous key exchange. Both sides do not need to know each other's public keys. The first client sends an initial random value to the server. The server receives a random number,

then it adds an additional number and sends it back to the client. The client computes the session key, encrypts it, and sends it to the server. After the reception and decryption of the session key, both sides have the same session key.

MonoCypher's key exchange system works a little bit differently. It needs the remote public key from another device and the local private key to exchange a session key using the Diffie-Hellman algorithm. The problem of this key exchange system is that it generates the same session keys unless local private key or remote public key changes. That means that session keys will be the same each time two drones will start an agreement.

To make this session key different, it is possible to generate random nonce with the length of the session key on both sides. Generated nonce needs to be sent to another device where the received and local nonce could be combined using the XOR function. This way, both devices have the same random nonce where each side had a fifty percent commitment. Random nonce during exchange might be public. When both devices have the same nonce, the generated session key might be XOR-ed with a random nonce. This way, the session key will be different each time the same devices will communicate with each other.

As it was already described to exchange session key and then XOR it, only the local public key and a random nonce need to be sent to the other side. To make the exchange system even more straightforward, this certificate message could be broadcasted, for instance every five seconds. This way, newly connected devices might get certificates and random nonces from all active devices during few seconds and without any request.

There is also a difference between those two libraries in the size of the final encrypted data after authenticated encryption. LibHydrogen adds 36 additional bytes (random nonce and authentication tag) to the final ciphertext pointer. That means that if the message has the size of 279 bytes, there will be no space for an additional 36 bytes in the payload part. Payload part is limited to 255 bytes. In the LibHydrogen API the final ciphertext with authentication tag is one pointer of type uint8_t and there is no way to split authentication tag and ciphertext.

MonoCypher library during encryption also needs a random nonce, which should be different for each message and place to store authentication tag. The difference is in memory storage, wherein the authentication tag is saved in a different pointer. This way, it is possible to preserve the maximum length of the payload part in the MAVLink 2.0 packet. To add an authentication tag and nonce into MAVLink 2.0 packet, additional space needs to be reserved. Using incompatibility flags, it is possible to inform the receiver that an additional 24 bytes with authentication tag and 16 bytes with random nonce will be transmitted.

4.3.2 Performance comparison

To find out which library has better performance, an example code was created. Every API function that was expected to be used in final implementation was executed on ARM and X86 architecture. The same data were used as input for both libraries. Performance results are based on usage of 100% of CPU performance. The main difference in performance tests was in the session key exchange system.

As it was described in the previous subsection, MonoCypher library generates a session key based on the remote public key and local private key. Message encryption was benchmarked on message with the size of 87 bytes. Results might be seen on Fig. 4.1 and 4.2

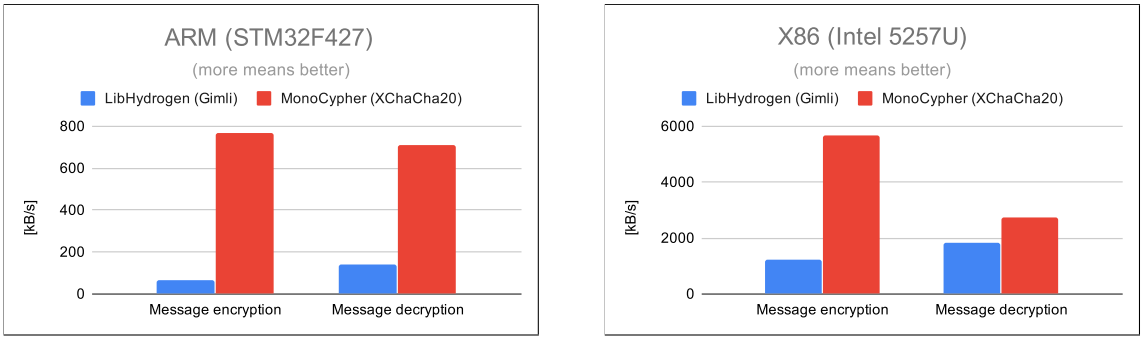


Fig. 4.1: Encryption performance of 87 bytes message, left - ARM, right - X86

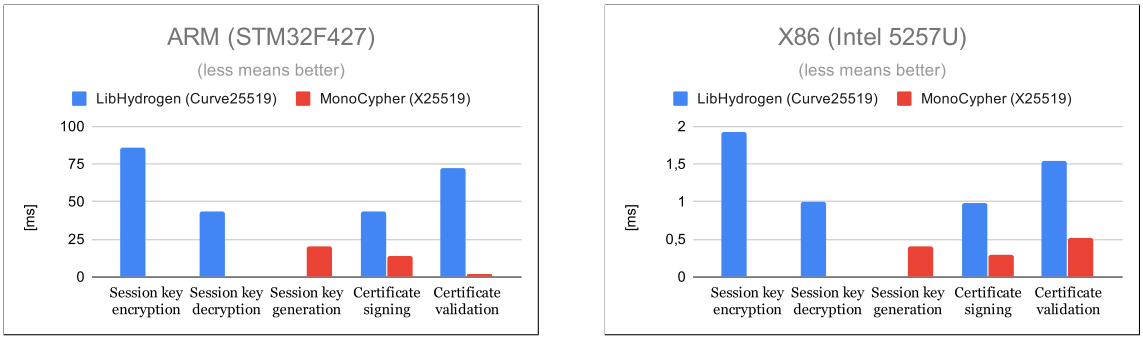


Fig. 4.2: Other tasks performance, left - ARM, right - X86

4.3.3 Conclusion

Both libraries were compared with regards to functionality and performance in order to find out which will be more suitable for security implementation in the PX4 platform. MonoCypher was significantly faster over LibHydrogen. For some tasks, MonoCypher speed was up to seventy times faster than LibHydrogen. Differences

between these two libraries are more considerable in performance on ARM based processor.

Also, the key exchange system in MonoCypher makes the final implementation into MAVLink 2.0 simpler because both sides compute session key based on provided remote public keys, local private key, and XORed random nonce. There is no handshake needed.

Based on the speed benchmark and its features, MonoCypher was chosen as the cryptography library for security implementation in PX4.

5 Random number generation on PX4 platform

After choosing an encryption library that would be compatible with any device that might use the MAVLink protocol, another crucial step is to create private keys and nonces with enough randomness. This means that keys and nonces bytes must be based on the sequence of random bits from the random generator with enough entropy. In that case, the attacker is not able to guess any part of the generated keys based on previous random values or using side-channel attacks.

There are two types of random generators:

- Pseudo-random generators,
- True random generators.

5.1 Pseudo-random number generators

Pseudorandom Number Generators (PRGN) as described in paper created by *National Institute of Standards and Technology* (NIST) organization [21] is a group of algorithms to generate numbers with nearly random properties deterministically. PRGN could be also called *Deterministic Random Number Generators* (DRGN).

The result of PRGN is always based on an input seed that must be inserted before any other random number generation. Input seed needs to be generated using a true random number generator. PRGN are mostly based on hash functions, where it is not possible to find out what the input number was based on the knowledge of the output number.

One of the most important features that the hash function should have is that there should be no connection between the input value and the output value. That means that after a few iterations with random input seed, there should be no way to find which value will be generated in next iteration. That does not apply to devices or persons that know the input seed and the number of iterations through a hash function.

Both libraries that were described in detail in the previous chapter use this method to generate pseudo-random numbers. They need only initial seed and then using the hash function like Gimli or Blake2b, they were able to create a pseudo-random numbers. These pseudo-random numbers are safe for cryptography only if initial seed has high entropy.

Examples of this seed source could be, for instance, user inputs on desktop PC or hard drive events. In the case of UAVs, it is possible to use output values from autopilot IMU. Those sensors generate an output signal that contains noise with a

high level of entropy. To get more information about these sources of entropy, it is possible to find more information in the 2016 paper by Kyle Wallace [22].

5.2 True-random number generators

As it was already said in the previous section, both libraries have implemented pseudo-random number generators, which need initial seed. To create these random numbers, we need true-random number generators. Such devices are mostly based on physical phenomena like thermal noise or photoelectric effect. These stochastic processes generate low-level noise. When this signal is converted into digital bits, it is possible to use them to create a random value.

To prove that *True Random Number Generator* (TRNG) is truly random, it must pass the tests of randomness. This includes tests like frequency test, where a number of zero bits and one bit should be the same in the block of bits. Another test is run test, where it measures how many bits with the same value are generated in a row. Using this test, it is possible to determine that the generator has enough entropy and that we can rely on them.

True random generators tests will be described more in the next chapters, where are results of random generators on x86 and ARM platforms.

5.2.1 True number generator on Pixhawk

Pixhawk 1 and CubePilot Black has onboard STM32F427 CPU. This processor has integrated TRNG, which is based on an analog circuit and allows us to generate 32-bit random numbers based on the analog noise [23]. This circuit is constructed from ring oscillators. Outputs from those oscillators are XORed with a dedicated clock. To understand how whole TRNG works on STM32F427 processor, it is possible to inspect Fig. 5.1. To enable this circuit in NUTTX OS, it was necessary to add CONFIG_STM32_RNG=y in the firmware defconfig file. After enabling TRNG it is possible to read random numbers from /dev/random device.

5.2.2 True number generator in QGroundControl

To generate safely random numbers in QGroundControl, Qt frameworks offer QRandomGenerator class. This high-level class is well documented on the Qt website [24]. Using a static public member system(), it is possible to call generate() function, which accesses systems cryptographically-safe random generators. Public member system() on Unix systems is reading values from /dev/urandom, or use system calls

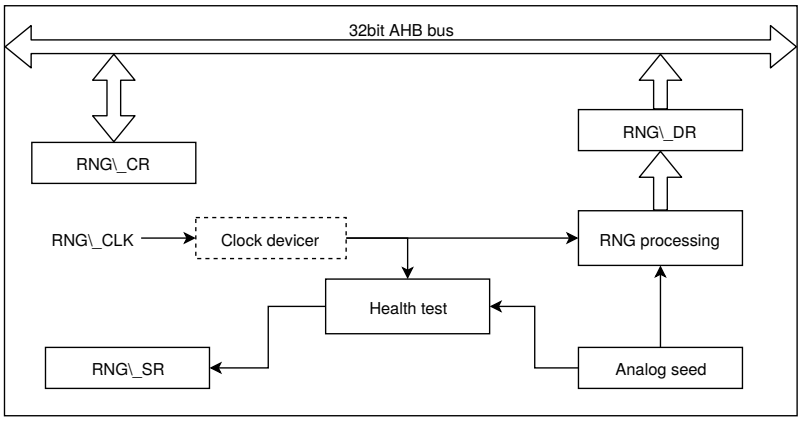


Fig. 5.1: Diagram of TRNG on STM32F427 [23]

getrandom() to get a random number. QRandomGnerator class supports most modern operating systems like macOS, Microsoft Windows, and Linux OS distributions.

5.3 NIST tests of randomness

To find out if TRNG are truly random, NIST created a group of statistical tests. This group of tests consists of 15 statistical tests where each of them searches different non-random binary sequences. For each test, there is a statistical p-value as output. P-value can range from 0 to 1. Parameter α, which also ranges from 0 to 1, indicates what percentage of sequences can be considered as non-random. If generator output values pass all tests near α value, it is considered as acceptable TRNG. The value of α depends on the application. In cryptography α is is usually around 0.01.

To test a huge number of zero and one bits, it needs to be divided into smaller groups. Each of these small groups need to go through all the tests, and if the resulting p-value of each test is bigger than α, then the tested group is considered random. In order to verify the hypothesis that sequences are random, p-values need to be equally distributed from 0 to 1 [21]. To test random numbers, the code made by GINARTeam [25] was used. Implementation of the test suite was written in the Python programming language. As can be seen on histograms in the appendix B, the distribution of p-values in all tests is good enough for both tested devices. The only results that look suspicious are from the Maurer's universal tests.

In the appendix B, the graphs on the left side plot the results of randomly generated sequences on Pixhawk. The graphs on the right side describe the results of tests based on random sequences generated on the X86 processor using the QRandomGenerator class from the Qt framework.

6 PX4 security vulnerabilities

As was described in the previous chapters, PX4 has no security layers that might protect the telemetry messages confidentiality and authenticity. Currently, there is an option to add message integrity protection by enabling message signing. The problem is that when using mission planner software QGroundControl, it is impossible to configure message signing keys. This means that information in any sent telemetry message is available or might be disclosed by an unauthorized person. Also, the message might be modified without any key, and the receiver can not find out where the message comes from.

The first part of this chapter will introduce the PX4 security vulnerabilities. The second part will present how algorithms implemented in the MonoCypher library might help in fixing them. For example, there were tests of attacks on communication between the control station (QGroundControl, Windows) and Pixhawk 1 autopilot. In these tests, devices communicate over SiK radios on 443 MHz. The attacker (QGroundControl, macOS) also uses SiK radio on the same frequency. All SiK radios work on the same channel so, if more than two devices are active, communication could be disturbed.

During the security tests, there was no documentation about the configuration of the signing system. It was implemented, but there was no easy way to start message signing.

6.1 Preparations

After starting up the control station and autopilot, both can communicate with each other. From GCS, it is possible to plan missions, change parameters, arm vehicle, or change flight modes. This provides full control over the UAV. At this point, an additional SiK radio is turned on and connected to the third computer. Because the attacker's computers also use QGroundControl software. During the next few seconds, all parameters are received, and the attacker has full access to UAV with the same privileges as his original maintainer.

6.2 Eavesdropping on MAVLink communication

Right after connecting into the vulnerable vehicle, it is possible to look into the MAVLink inspector, where all the newest messages are stored. It is possible to view the current position, attitude, airspeed, or altitude of the vehicle. During this part of the connection, QGroundControl draws the vehicle's flight path based on received messages. From this flight path, it is possible to guess the next route. The another

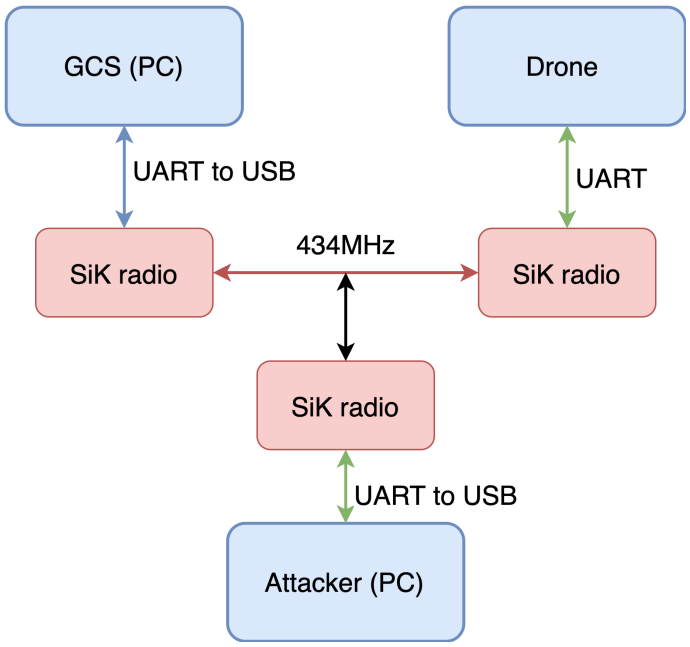


Fig. 6.1: Diagram of attack scenario

information that might be confidential is the flight mission that is drawn into the QGroundControl map after downloading all parameters.

6.3 Shut down of vehicle during flight

When QGroundControl is connected to the unmanned vehicle, there is a possibility to disarm the vehicle without any validation. After pressing the Arm button, a small slider appears at the bottom of QGroundControl. If UAV is armed and landed, then there is no warning, and right after activation of the slider, the vehicle disables all PWM outputs that control motors or servos. If the UAV is flying, then the small slider appears at the bottom of QGroundControl with a little warning. If the slider is activated, PWM outputs are disabled and motors stop to rotate. All connected servos are also disabled.

6.4 Changing vehicles flight mission

During the time when QGroundControl is connected, it is possible to create an autonomous mission in the mission planner tab and send an entire mission to the connected vehicle. To send the mission using the MAVLink protocol to UAV, the ground control station needs to send MISSION_COUNT message, which gives information to the UAV about how many mission items need to be downloaded. Then

the UAV sends MISSION_REQUEST_INT commands to request each mission item separately.

If the attacker wants to upload a new mission, it just needs to send the MISSION_COUNT message. The UAV will start to send requests for new mission items. After all mission items are received, QGroundControl asks the attacker if he wants to fly a new mission. After confirmation, the plane will start to fly through the new mission.

6.5 Change of vehicle parameters

During a flight, it is possible to change flight parameters. This vulnerability could have been created as a feature to improve PID tuning, because thanks to it it is possible to tune the whole plane or multi-copter during a few test flights. On the other hand, it gives the attacker a tool to make an attitude or position control very unstable. This might result in a crash. The flight logs contain no information about any change of parameters during the flight, causing the possible investigation of the crash to be complicated.

6.6 Taking control of the vehicle

In the MAVLink protocol, there is a message type RC_CHANNELS_OVERRIDE. Messages of this type override received PPM signal. Original PPM signal is received from the RC transmitter and send to a PWM rail. RC_CHANNELS_OVERRIDE message allows for control of the plane similar to how it would be if it received a PPM signal from the original RC transmitter, even when there is an active real RC transmitter that sends the true PPM signal. Once the attacker takes control, there is no way for the original user to override these messages because they have higher priority. The attacker is then able to fly with UAV without any limitations in the telemetry range of radio connection.

6.7 Summary

If any user uses SiK radio, MAVLink Wi-Fi module, or other radios without communication channel encryption, then their vehicles might be hacked, stolen, or might crash without any clue. The only way to detect attackers is to watch if any malicious messages appear in QGroundControl in the MAVLink inspector tab, where all the broadcasted messages in the network are shown.

Tab. 6.1: Identified risks of PX4 firmware on application layer

Name	Risk	Process	Countermeasures
Eavesdropping	Confidential information might be revealed	After connecting to vehicle using telemetry radio, it is possible to request all data from vehicle	Message encryption
Vehicle shut down	Vehicle during flight might be destroyed	QGC connects to vehicle, disarme vehicle during flight	Message signing
Change of flight mission	Vehicle might be stolen during autonomous flight	QGC connects to vehicle, sends new flight mission	Message signing
Change of parameters	Vehicle might be destroyed after flight parameters are changed	QGC connects to vehicle, changes flight parameters	Message signing
Take control	vehicle might be stolen during autonomous flight	Sending RC_OVERRIDE messages to vehicle over telemetry radio	Message signing

To remove all these vulnerabilities, it is necessary to have at least a whole network communication encrypted using an symmetric encryption algorithm. That solves described problems, but in a giant network, it brings additional key management. In the next chapter, a new secure communication system will be introduced and then implemented.

7 PX4 security architecture

This chapter describes the design of a new security implementation in more detail. All the ideas mentioned in this chapter have been discussed with the developers of the PX4 platform to achieve the highest possible compatibility with current and next versions of the MAVLink protocol.

7.1 Creating and signing vehicles certificate

Before any system in the network is able to communicate with other devices securely, a person that maintains the fleet of all devices must create the authority signing keys. With these keys, it is possible to sign device certificates so that they may be trusted.

The next step is to create individual certificates for all devices. Each of them needs to have its device's name, maintainer, privilege level, the public key of the device, and a signed hash. The input of the hash function is all the previous parts of the certificate. The resulting hash is then signed using the private authority key.

If an attacker attempts to modify the certificate, the final hash would be different than the one signed after being decrypted using the authority public key. After the whole certificate is created, it is saved into a file with the public key of the authority.

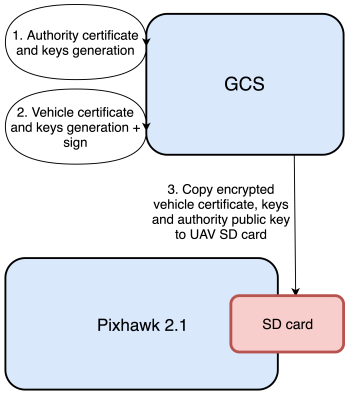


Fig. 7.1: Diagram of key generation

The first problem that must be dealt with is how to safely store the whole certificate. One way might be by saving all the data into a TPM chip. Another place to save a certificate containing the authority public key might be an SD card, which should be installed into every Pixhawk.

The problem is that an SD card might be removed and the certificate could be copied to another device. To prevent this, all data stored in the certificate file should be encrypted with a symmetric secret key.

In case of the QGroundControl software, the certificate is stored on a local hard drive. In both cases, an asymmetric secret key for certificate decryption should be stored in the firmware and QGroundControl source code as a variable or constant. After the SD card is inserted into the autopilot board with all the necessary and encrypted data, autopilot is ready to be powered on. Key generation process is shown in figure 7.1.

7.2 Diagram of key generation

After powering on, the autopilot reads the file with certificate and key exchange keys. If the certificate is encrypted, then the autopilot decrypts it using secret key pre-baked in the source code. If a certificate is stored in plain text, autopilot reads it without the decryption process, loading all data into memory.

The MAVLink module is then started. With messages like HEARTBEAT or RADIO_STATUS, it also starts to broadcast the newly designed message CERTIFICATE. That allows a remote device to verify this autopilot using the authority signing the public key.

7.3 Validation of certificate and public key

To validate other devices' certificates, every device has an authority signing public key in their certificate file on the SD card. After receiving a CERTIFICATE message from another device, verification needs to be done. To do all parts of CERTIFICATE messages, like name, maintainer, privilege number and key exchange, the public key must be hashed. The signed hash should be then decrypted and compared with locally created hash. If they are the same, then the provided certificate is valid, and it means that the key exchange public key can be used to generate a session key.

7.4 Session key agreement

In MonoCypher, it is possible to generate a session key based on the public key of the remote device and local private key. This way, it is possible to generate the same session key on both sides. For every session, this key will be the same after multiple generations, because the key exchange is based on the Diffie-Hellman algorithm.

To make different session key, the certificate message also has a random nonce which is generated after the start of the device. Once both nonces are known on all sides, they need to be XORed. That way, both devices on each side should have the

same random nonce. Then they can XOR the session key with the random nonce. That enables having different session keys for every session.

7.5 Message encryption

After both sides have the same session key, it is possible to encrypt all messages and commands. An exception to this rule are messages that might be publicly broadcasted, such as HEARTBEAT or CERTIFICATE. Every other message will be encrypted using the session key. To make encryption design compatible with MAVLink 2.0, only the payload part will be encrypted without any additional bytes. The functions used for encryption and decryption in the MonoCypher libraries need three pointers as their parameters: key, nonce and mac. As it was already described in the previous section, the key is generated between both sides, and then generated key is XORed with result of XOR operation on remote and local certificate random nonces.

Another nonce is used to make the same messages after encryption to appear different. Because this nonce must be the same on both sides and there is no space left in standard MAVLink 2.0 packet to send it to another device, some part of the MAVLink header needs to be extended. By using two new incompatible flags in MAVLink 2.0. it is possible to send 16 bytes for mac and 24 for the nonce. During the receiving process, when incompatibility byte is received, and there is an active MAVLINK_IFLAG_ENCRYPTED flag (0x02), the receiver will wait for additional 40 bytes after the CRC part. If encryption with random nonce is done and mac is filled, the whole packet with those 40 additional bytes may be sent.

During the receiving process, it is possible to verify that the message wasn't modified. MonoCypher crypto_lock() verifies if the provided mac from the MAVLink packet is valid for the decryption process. If the mac or the message have been modified, crypto_lock() function returns -1 value. This way, it is possible to verify that the message could only be created by the owner of the symmetric key. This system replaces the signing system.

7.6 Addressing system

Because the MAVLink packet is missing destination address (System ID), it is not possible to find out if the received message should be decrypted, because it could be encrypted using a different session key. This situation may occur if more than two devices are on the same network.

To solve this problem, an additional byte with destination address should be added behind nonce when the MAVLINK_IFLAG_ENCRYPT flag is activated. This way, it is possible to choose symmetric key based on the source address and decrypt only those packets that are intended for to this local device only.

Without this addressing, each received message needs to be decrypted and verified. This will cause undesirable CPU load. Addressing system will also allow the implementation of an access control system. If a device that receives encrypted message knows the source address, it can find the privilege level between saved certificates of other devices. A certificate of a remote device might only be saved when the signature is valid.

7.7 Example of security implementation

To demonstrate how the security implementation should work, an example has been created based on the design described in the previous chapter. There is a short source code [26] provided as an example for each reviewed library that shows how the final communication system should work.

This example was designed before choosing MonoCypher as a library for the final implementation. The example is mainly oriented to demonstrate how messages should be encrypted and signed and how key exchange should work. Also, it is possible to see there idea of certificate signing and its verification.

After looking at this example code a person with a basic knowledge of cryptography should understand what was meant and should be able to imagine how the newly secured communication will work. An example code was written for both platforms. This example code was used to measure the performance of both libraries. Figure 7.2 shows idea of communication with new implementation.

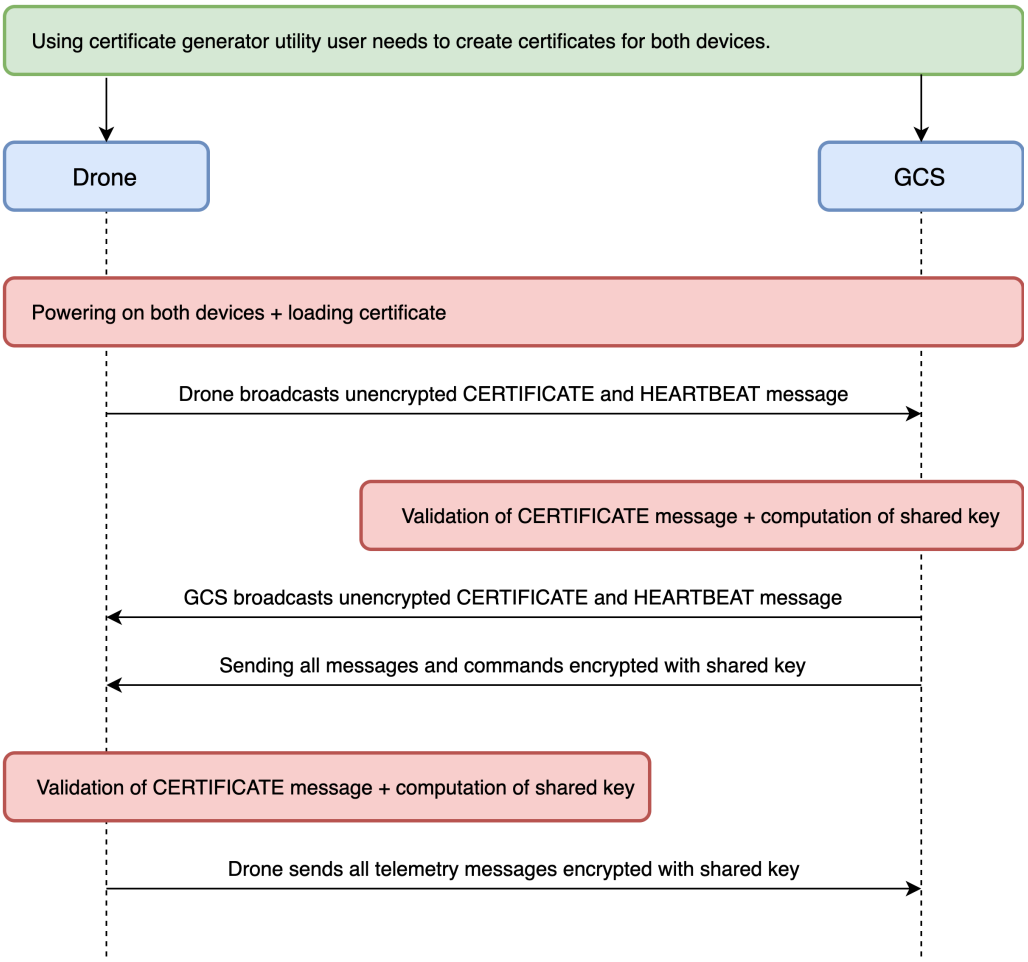


Fig. 7.2: Diagram of key exchange system

8 Final implementation

This chapter will show how the final implementation was created and which changes in source code were done. The whole implementation is divided into four parts based on GitHub public repositories. These repositories include: mavlink/mavlink [27], ArduPilot/pymavlink [28], PX4/Firmware [29] and mavlink/qgroundcontrol [30].

Each of these repositories was cloned on a local disk and modified to add desired functionality. The implementation currently contains a new key exchange system, message signing, and message encryption. It still lacks certificate encryption, the destination address in the MAVLink packet, access control system, and support for encryption of more than two devices in a network. These things still need to be discussed with the PX4 community to maintain compatibility.

8.1 Mavlink repository

This repository contains all the message definition divided by dialects. The most used dialect is the "common" dialect. Definitions of these messages are stored in the XML file. To add the CERTIFICATE message, the file common.xml needs to be modified.

The final version of the CERTIFICATE message in XML format that has been used created is shown in the appendix A. This repository was pull requested with the newly added CERTIFICATE message. Part of the mavlink repository is also a submodule with a pymavlink repository. Because this repository was also modified, it is necessary to checkout commit with the newly implemented security system.

The changes made to the mavlink repository are visible on GitHub in rligocki/-mavlink [31] repository. This repository already has an active pull request waiting to be accepted.

8.2 Pymavlink repository

Part of mavlink/mavlink is ArduPilot/pymavlink repository. This python source code is a python implementation of MAVLink. It is also designed to generate the MAVLink library with all helper functions and messages in one supported programming language. To add security implementation into PX4 firmware and QGroundControl, it was necessary to modify files in pymavlink/generator/C/include_v2.0/ folder.

These files contain helper functions in C language. Their purpose is to put together, prepare to send, receive, and parse the mavlink packet. Functions in

mavlink_helpers.h file were modified so they can activate the encryption incompatibility flag, encrypt data and add the nonce and mac parts. Another important part of the new implementation that was changed in mavlink_helpers.h is a way to handle received bytes and to put the mavlink packet back together.

If encryption incompatibility flag is active, additional 40 bytes are received. After the whole packet is reconstructed, encrypted data might be decrypted. If mac, nonce, or encrypted data are corrupted, the decryption function will return -1 value. Most of the additional code was added into the mavlink_helper.h and mavlink_types.h files. New packet format can be viewed on Fig. 8.1.

MonoCypher library needs to be modified to work as a single .h file. That means it is necessary to insert full source code from .c file to .h declarations. Pymavlink generator supports both python2 and python3. All changes in pymavlink are visible in rligocki/pymavlink [32] repository. Rligocki/pymavlink also has an active pull request into ardupilot/pymavlink.

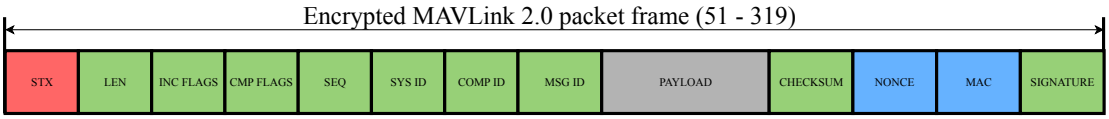


Fig. 8.1: Encrypted MAVLink 2.0 packet frame

8.3 Firmware and QGroundControl repositories

In these repositories, it was necessary to update mavlink libraries generated from the latest pymavlink repository commit. In the case of both software, it was necessary to add functions that set nonces for the certificate at the beginning of runtime and also set the random nonce for each message.

Another step that needs to be done was to read the certificate file and turn on CERTIFICATE message broadcasting. The last step was to add CERTIFICATE message handling after its successful reception. All changes are visible in latest commit in rligocki/Firmware [33] and rligocki/QGroundControl [34] repositories.

To prove how final implementation works, an example message sent from PX4 to QGroundControl was captured before and after it was encrypted. It is possible to view the original and encrypted ATTITUDE message on table 8.1. In the header field containing incompatibility, the flag was modified. The payload part has been encrypted. After payload was encrypted, CRC checksum, a random nonce, and mac are inserted into the footer of the packet.

Tab. 8.1: Example of MAVLink message encryption

MAVLink packets		Plaintext [Hex]	Ciphertext [Hex]
Header	mag	fd	fd
	len	1c	1c
	inc flags	00	02
	com flags	00	00
	seq	17	17
	sys id	01	01
	comp id	01	01
	msg id	1e0000	1e0000
Payload		ee6b0000fa2ddebcbbf298bdf1fce b3f80acd43a80782a3b0ce7c5bb	7584f86fad3c750e32672d25fdea8 8f4b720b8c0d6d4bb31eb6a4789
CRC		64d0	afba
Footer	Mac	00000000000000000000000000000 00000	5dfae4a465297a1b6ea3939073c 8a6ec
	Nonce	00000000000000000000000000 000000000000000000	f916f9cafb213d6e72c5196eca874 63168e03a1723395313

8.4 TCP/IP compatibility

By default the MAVLink packet can be encapsulated into a TCP/IP packet. The whole packet is inserted into the data part of the IPv4 packet. With destination and source address, it is possible to route these packets through a TCP/IP network.

With new security implementation, the only difference is in the length of the packet inserted into the data part of the TCP/IP packet. This way, compatibility with TCP/IP is broken. To get it working, devices that encapsulate TCP/IP communication must be compiled with the modified MAVLink library. This new version knows how to handle MAVLink packets with additional bytes and also has implemented a CERTIFICATE message. Change of MAVLink library need to be done in GitHub repositories like dogmaphobic/mavesp8266 or ArduPilot/MAVProxy.

8.5 Additional traffic

UAVs mostly use telemetry radios with long-range, but with throughput of only about 64kb/s by default. In this case, it is essential to review the size of additional traffic that is caused by the new security implementation. The difference in messages length is visible in appendix C. The implementation on average adds about 90% of the original length. That means that update rates of messages on a fully used communication channel will be cut in half.

To lower the data transfer requirements of the new security implementation, the MonoCypher library needs to be modified to lower the size of mac. Another way to reduce data requirements is to use a smaller nonce. In MonoCypher, nonce has 24 bytes length. If nonce was to be reduced to 8 bytes, then the traffic of the nonce part will be three times smaller. To fulfill 24 bytes fixed nonce length, new 8 bytes nonce could be used three times.

These changes could be proposed to community developers and then it might be possible to find out if these changes should be applied or current implementation might stay as it is. After entering the "mavlink status" command into the original PX4 console, USB connection uses about 25 kB/s in TX. On the modified version of the PX4 console, TX was about 37 kB/s. It means that communication traffic is about 48% larger.

8.6 Performance and consumption

To find out how the final implementation changes CPU usage, it is possible to run console "top" service on PX4 firmware. In the modified version with the autopilot idle, the average CPU usage was about 80 percent. Mavlink_if0 service that maintains the USB high throughput communication channel used 20 percent of the CPU. In the original version of the firmware, average CPU usage was about 65 percent. In the case of mavlink_if0 service, CPU usage was 9 percent. It means that in case of modified version of mavlink_if0 service CPU usage has two times bigger. To improve the performance of final implementation, it is recommended to lower the maximum data throughput of the USB MAVLink connection or to disable encryption.

If an attacker has access to the physical hardware of the autopilot, he can upload any modified version of firmware or read private flight parameters from EEPROM. That means that the encryption of the USB channel is currently useless. By default the TELEM1 port, which is used to connect pixhawk autopilot to telemetry radio, has a limited default throughput of 1.2kB/s. That means that the usage of CPU could be much lower if the communication encryption is applied only on the TELEM1 port connection. This way, it is possible to achieve a secure communication channel, but also good performance.

To find out if the final implementation may increase current consumption and if it has some influence on battery life, it was necessary to measure autopilot current consumption with the new implementation and without it. To measure current consumption, it was necessary to power pixhawk by circuit as Rp resistor. With R1 that had 5.1 Ω, it was possible to measure a voltage drop of R1 in the whole circuit.

The value of R1 was measured with Metex M-3890D. To measure the voltage drop on the R1 resistor the OWON SmartDS8102 oscilloscope was used. In appendix D it is possible to view the results obtained from this oscilloscope. It is also possible to see power sources (MATRIX MPS-3003L-3) output voltage. In both measurements, the maximum, minimum, and average value were captured. The formula 8.1 was used for computation of average current consumption.

$$I = U_R/R_1 \tag{8.1}$$

The results are 0.1722 A with new security implementation and 0.1758 A without security implementation. These results signify that the that the new implementation does not affect current consumption. The difference between these results is the consumption of other circuits on the autopilot board.

The whole implementation affects only the CPU load. To improve current measurement accuracy it is necessary to measure CPU using separate power sources. This way there will be no other devices consumption included.

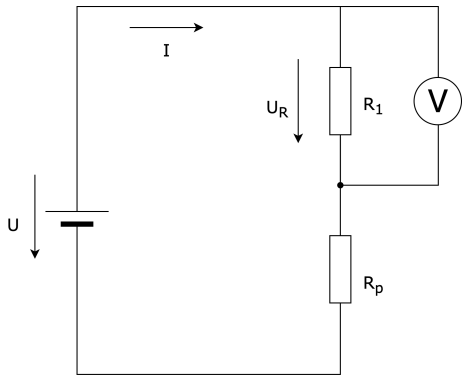


Fig. 8.2: Connection of Pixhawk (Rp) to measure current consumption

8.7 Future plans

The final implementation provides secure communication when in the MAVLink network only if there are two devices. Currently, there is no support to store multiple certificates and multiple symmetric keys. To add support for multiple devices, there is a need to add optional destination address byte in the MAVLink packet.

This will allow the receiver to identify to whom the packet is intended. If it is intended to a receiver, the packet might be decrypted with a symmetric key. Support for multiple devices will lower the update rate of MAVLink messages. The reason is that with the same throughput on the radio channel, it is necessary to encrypt each message multiple times with different symmetric keys. If there are four other

devices in the MAVLink network, for instance, the ATTITUDE message needs to be encrypted four times and also sent four times. Encryption of this message needs to use a different key each time.

One of the author's private goals was to achieve a pull request of final implementation into the master branch of the PX4 software stack. That means that this thesis will be used as proof to show that this final implementation can be merged to the master branch of the PX4 repository. If this implementation is merged and modified to make it usable for non-developer users, the author will start to work on support for multiple devices in the MAVLink network.

Conclusion

The main goal of this thesis was to analyze the security of the PX4 platform and then propose a new solution fixing the security vulnerabilities discovered in the analysis. To do this, first the hardware and software of PX4 platform was analyzed in detail. The software part of the PX4 platform includes autopilot firmware, ground control station software QGroundControl, MAVLink protocol. The hardware part introduces the autopilot board and compatible telemetry radios.

Based on a security focused analysis, a lack of message encryption and signing were determined as the principal vulnerabilities of the PX4 platform. With more profound knowledge of PX4 software, hardware, and MAVLink protocol, it was possible to choose a suitable C cryptography library. The MonoCypher [20] library was chosen as the main cryptography library.

This library allows us to encrypt and sign messages, but also provides a solution to exchange symmetric keys using the insecure communication channel. These features add the possibility to make the exchange of MAVLink messages more secure. The whole implementation may be realized With only a few minor changes in the header of the MAVLink packet and by adding a MAC and nonce part to the footer.

In order to find out if the Pixhawk hardware has enough performance to encrypt and decrypt messages, it was necessary to perform a benchmark. Results showed that Pixhawk 1 could encrypt data with the speed up to 800 kB per second.

Another part of the thesis was to verify if PX4 hardware has safe TRNG. To do this, tests on the NIST test suit have been done for the ARM and the X86 platform. Results showed that autopilot boards based on the STM32F4 processor generate sequences of bits with enough entropy. QGroundControl, based on the Qt framework, was also able to generate random numbers with enough entropy. That means that all principal parts of the PX4 platform were able to generate keys for encryption and decryption safely and had enough performance.

In the next part, a new security implementation was created and presented. It was necessary to create an implementation for each of the used repositories. Most of the work was done in the pymavlink repository. This repository contained helper functions to handle MAVLink communication on autopilot and ground control station. With a modified pymavlink repository, it is possible to add new functionality into PX4 firmware and QGroundControl, where pymavlink stands as a dependency library.

After new functionality was added, it was necessary to prove its functionality. An example of an encrypted packet, performance, and consumption tests show that the new security implementation works. Currently, the author is modifying repositories to merge the whole final implementation to the public repositories master branch.

Bibliography

[1] DroneDeploy. Commercial drone industry trends, 2018. URL: https://dronedeploy-www.cdn.prismic.io/dronedeploy-www%2Fae535fda-dfc9-4bcf-9743-292df714e9fe_dd__2018_trends_report-f.pdf.

[2] Adarsh Gangwal. Blood delivery by drones: A case study on zipline. http://www.ijirset.com/upload/2019/august/63_Blood.PDF, 2019.

[3] Lorenz Meier. How i accidentally created the most used standards in the drone industry, 2019. URL: https://auterion.com/the-history-of-pixhawk/.

[4] PX4. Px4 user guide. https://github.com/PX4/px4_user_guide, 2019.

[5] ArduPilot Dev Team. Ardupilot documentation, 2019. URL: https://ardupilot.org/ardupilot/index.html.

[6] Shawn Herrick. What's the difference between a drone, uav and uas?, 2017. URL: https://www.identifiedtech.com/blog/uav-surveying/drone-technology-ending-the-drone-vs-uav-debate-drone-basics-101/.

[7] *Open Source Licensing*, 2004. URL: http://dl4a.org/uploads/pdf/ebooksclub.org__Open_Source_Licensing___Software_Freedom_and_Intellectual_Property_Law.pdf.

[8] DroneCode. Px4 architectural overview, 2019. URL: https://dev.px4.io/master/en/concept/architecture.html.

[9] mavlink. Cross-platform ground control station for drones (android, ios, mac os, linux, windows). https://github.com/mavlink/qgroundcontrol, 2019.

[10] DroneCode. Flight controller selection, 2019. URL: https://docs.px4.io/v1.9.0/en/getting_started/flight_controller_selection.html.

[11] 3D Robotics, 1608 Fourth Street Berkeley, CA 94710. *PIXHAWK AUTOPILOT - QUICK START GUIDE*, 2014. URL: https://3dr.com/wp-content/uploads/2017/03/pixhawk-manual-rev7-1.pdf.

[12] Hex Technology, Hong Kong. *PIXHAWK 2 AUTOPILOT - QUICK START GUIDE*, 2016. URL: http://www.hex.aero/wp-content/uploads/2016/09/PIXHAWK2-Assembly-Guide.pdf.

[13] Hex Technology, Hong Kong. *Pixhawk v2 Feature Overview*, 2016. URL: http://www.hex.aero/wp-content/uploads/2016/07/DRS_Pixhawk-2-17th-march-2016.pdf.

[14] Czech Telecommunication Office, Praha. *Všeobecné oprávnění č. VO-R/10/01.2019-1 k využívání rádiových kmitočtů a k provozování zařízení krátkého dosahu.*, 2019. URL: https://www.ctu.cz/sites/default/files/obsah/ctu/vseobecne-opravneni-c.vo-r/10/01.2019-1/obrazky/vo-r10-012019-1.pdf.

[15] ArduPilot. Tools and firmware for the si1000. https://github.com/ArduPilot/SiK, 2018.

[16] RFDesign Pty Ltd, 6/97 Jijaws Street Sumner Park, QLD 4074. *RFD900 Radio Modem Data Sheet*, 2013. URL: http://files.rfdesign.com.au/Files/documents/RFD900%20DataSheet.pdf.

[17] Hex Technology, Hong Kong. *HereLink - User manual*. URL: https://fccid.io/2ARLU-HA06071/User-Manual/User-Manual-4389016.pdf.

[18] mavlink. Mavlink developer guide. https://github.com/mavlink/mavlink-devguide, 2019.

[19] jedisct1. A lightweight, secure, easy-to-use crypto library suitable for constrained environments. https://github.com/jedisct1/libhydrogen, 2019.

[20] LoupVaillant. An easy to use, easy to deploy crypto library. https://github.com/LoupVaillant/Monocypher, 2019.

[21] National Institute of Standards and Technology, Gaithersburg. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 2010. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf.

[22] Kyle Wallace. Toward sensor-based random number generation for mobile and iot devices, 2016. URL: https://static1.squarespace.com/static/53065911e4b0cca0183fc14a/t/57fabdaa6a496306c83ca9a5/1476050348493/SensoRNG-preprint.pdf.

[23] STMicroelectronics. *STM32 microcontroller random number generation validation using the NIST statistical test suite*, 2019. URL: https://www.st.com/content/ccc/resource/technical/document/application_note/4a/6a/82/05/8e/9e/4e/94/DM00073853.pdf/files/DM00073853.pdf/jcr:content/translations/en.DM00073853.pdf.

[24] The Qt Company Ltd. *QRandomGenerator Class*, 2019. URL: https://doc.qt.io/qt-5/qrandomgenerator.html.

[25] GINARTeam. Nist's statistical test suite for random number generator (rng) that apply to ginar rng. https://github.com/GINARTeam/NIST-statistical-test, 2019.

[26] rligocki. Px4_example. https://github.com/rligocki/px4_example, 2020.

[27] mavlink. Mavlink. https://github.com/mavlink/mavlink, 2020.

[28] ArduPilot. python mavlink interface and utilities. https://github.com/ArduPilot/pymavlink, 2020.

[29] PX4. Px4 autopilot software. https://github.com/PX4/Firmware, 2020.

[30] mavlink. Qgroundcontrol ground control station. https://github.com/mavlink/qgroundcontrol, 2020.

[31] rliocki. Marshalling / communication library for drones. https://github.com/rligocki/mavlink, 2020.

[32] rligocki. python mavlink interface and utilities. https://github.com/rligocki/pymavlink, 2020.

[33] rligocki. Px4 pro autopilot software. https://github.com/rligocki/Firmware, 2020.

[34] rligocki. Qgroundcontrol ground control station. https://github.com/rligocki/qgroundcontrol, 2020.

[35] ArduPilot. Arduplane, arducopter, ardurover source. https://github.com/ArduPilot/ardupilot, 2019.

List of symbols, physical constants and abbreviations

AES Advanced Encryption Standard
AES-CCMP Advanced Encryption Standart-Counter Cipher Mode with Block Chaining Message Authentication Code Protocol
API Application Programming Interface
ARM Advanced RISC Machine
CAN Controller Area Network
CTO Czech Telecommunication Office
CPU Central Processing Unit
DIY Do It Yourself
DRGN Deterministic Random Number Generators
EEPROM Electrically erasable programmable read-only memory
ESC Electronic Speed Control
FMU Flight Management Unit
FMUv2 Flight Management Unit version 2
FMUv3 Flight Management Unit version 3
FPU Floating Point Unit
FPV First Person View
GCS Ground Control Station
GMT Greenwich Mean Time
GPL General Public License
GPS Global Positioning System
HMAC Keyed-Hashing for Message Authentication
HMI Human Machine Interface
ISC Internet Software Consortium
IEEE Institute of Electrical and Electronics Engineers
IMU Inertial Measurement Unit
IoT Internet of things
LGPLv3 Lesser General Public License version 3
MAVLink Micro Aerial Vehicle Link
MPU Magnetic Pickup Unit
MPU Magnetic Pickup Unit
NIST National Institute of Standards and Technology
OS Operation System
PCB Printed Circuit Board
PPM Pulse Phase Modulation

PRGN Pseudorandom Number Generators
PWM Pulse Width Modulation
PX4 Name of software flight stack, PX is shortage for Pixhawk, which was the name of the development team that created PX4. Number 4 means fourth rewrite of the PX flight control software
RAM Random-Access-Memory
RC Radio Controlled
RSSI Received Signal Strength Indication
RTOS Real Time Operation System
SD Secure Digital
SHA Secure Hash Algorithm
SoC System on a Chip
TPM Trusted platform module
TRNG True Random Number Generator
UART Universal Asynchronous Receiver-Transmitter
UAV Unmanned Aerial Vehicle
UGV Unmanned Ground Vehicle
USB Universal Serial Bus
USD United States Dollar
VTOL Vertical Takeoff and Land
WLAN Wireless Local Area Network
WPA2 Wi-Fi Protected Access 2
XML Extensible Markup Language

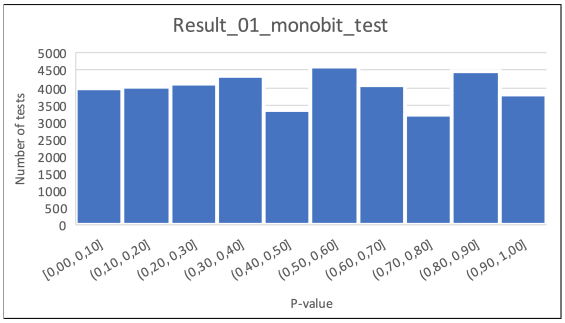
List of appendices

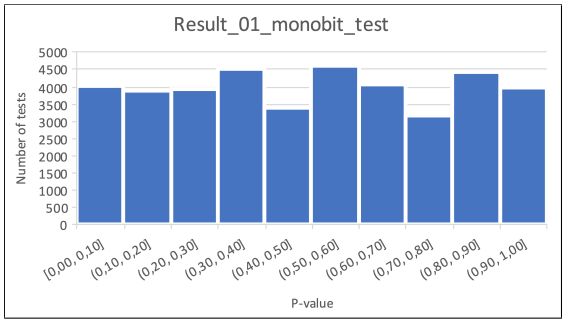
A Definition of CERTIFICATE message in XML format

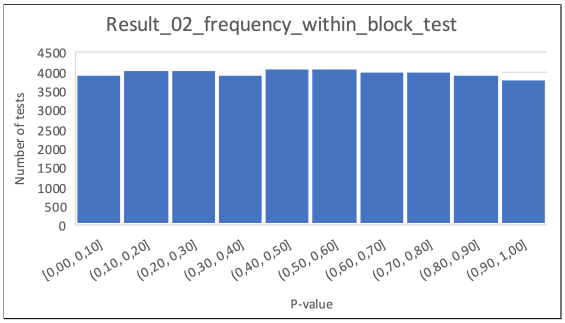
Listing A.1: Certificate stucture in C langugage

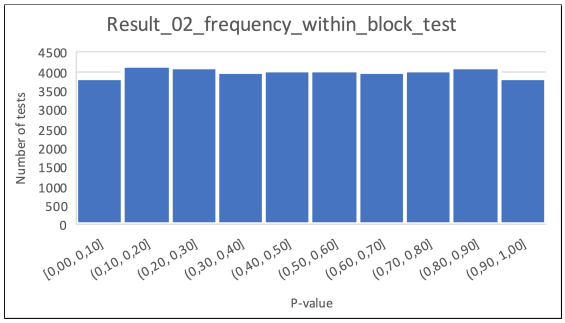
```
<message id="1000" name="CERTIFICATE">
<description>Signed certificate by authority, that is meant to
validate another device public key and privileges. If
certificate is trusted, then it is possible to communicate
with other device</description>
    <field type="uint8_t" name="device_id">
        Network address</field>
    <field type="char[20]" name="device_name">Name of
        machine</field>
    <field type="char[20]" name="maintainer">Name of
        maintainer</field>
    <field type="uint8_t" name="privileges">
        Privilages of device</field>
    <field type="uint8_t[32]" name="public_key">Public
        key of device</field>
    <field type="uint8_t[32]" name="nonce">
        Random nonce to generate different key each time
        </field>
    <field type="uint8_t[64]" name="sign">Hash
        of previous fields signed by authority </field>
</message>
```

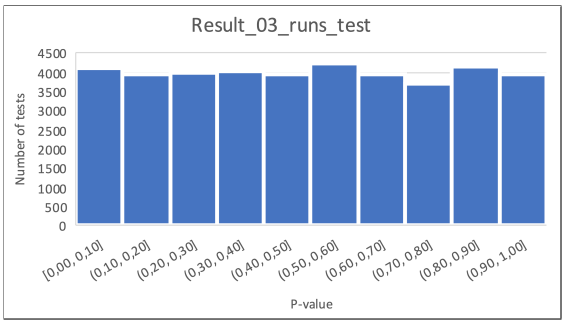
B Results of NIST tests

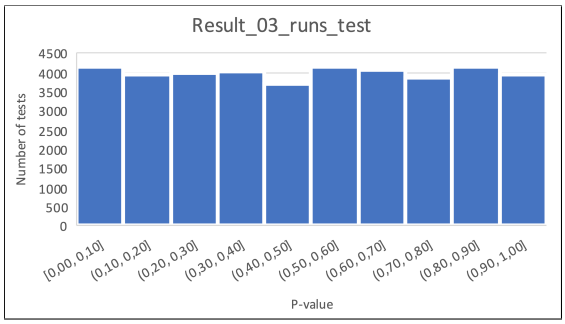


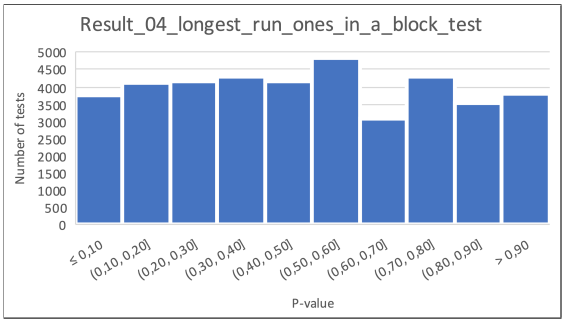


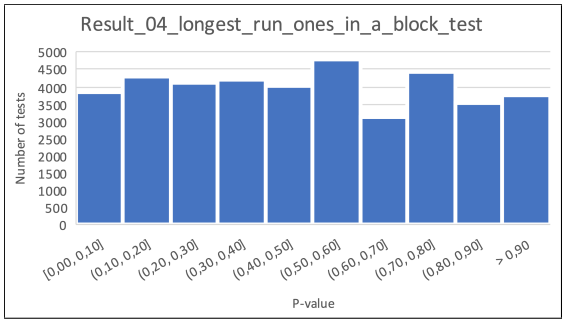


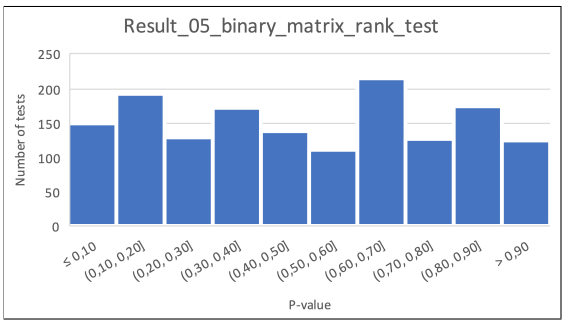


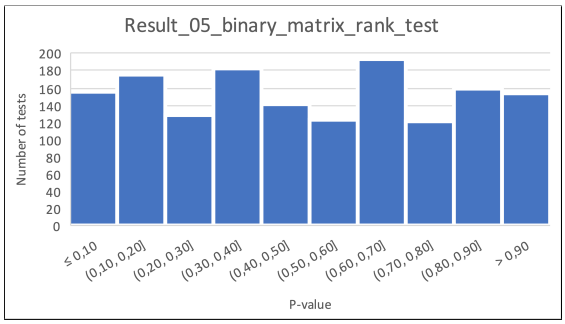


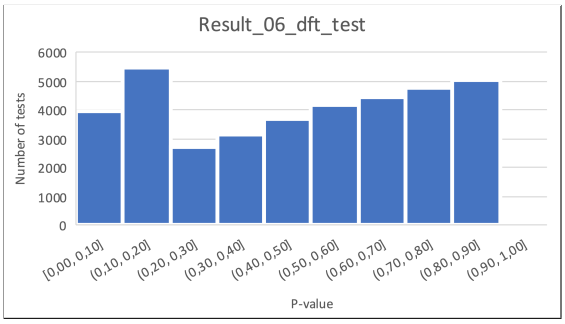


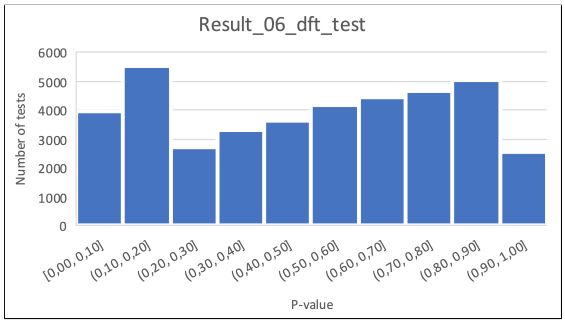

Result_04_longest_run_ones_in_a_block_test
Number of tests
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
≤ 0,10
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
> 0,90
P-value

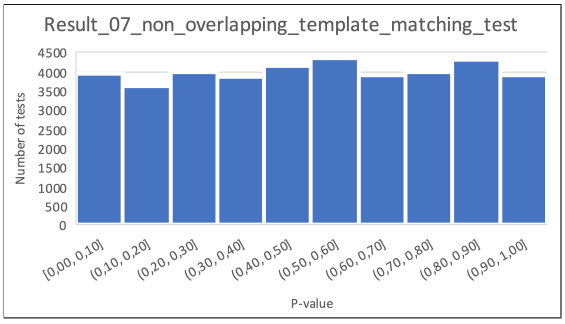

Result_04_longest_run_ones_in_a_block_test
Number of tests
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
[0,00, 0,10]
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
> 0,90
P-value

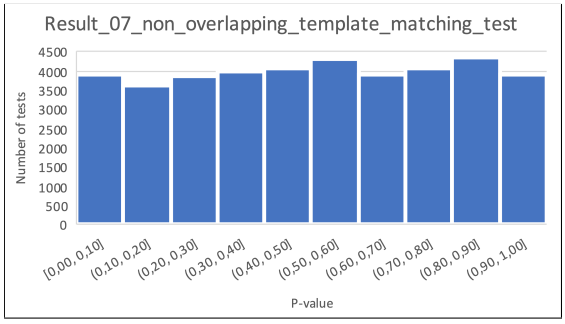

Result_05_binary_matrix_rank_test
Number of tests
250
200
150
100
50
0
≤ 0,10
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
> 0,90
P-value

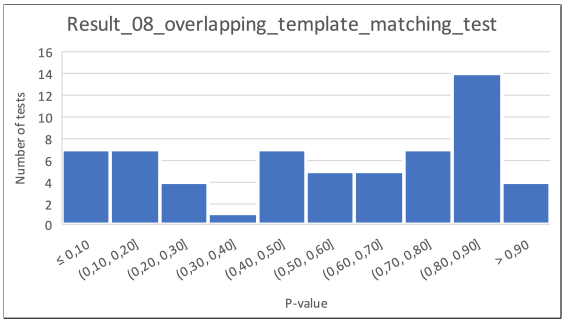

Result_05_binary_matrix_rank_test
Number of tests
200
180
160
140
120
100
80
60
40
20
0
≤ 0,10
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
> 0,90
P-value

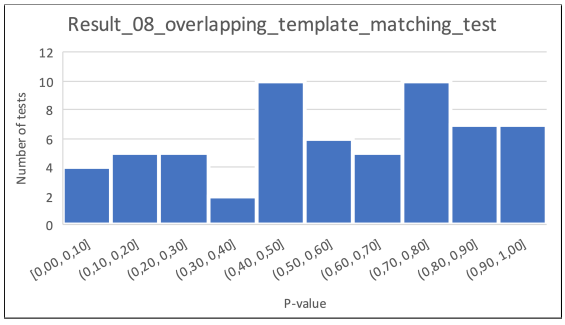

Result_06_dft_test
Number of tests
6000
5000
4000
3000
2000
1000
0
[0,00, 0,10]
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
(0,90, 1,00]
P-value

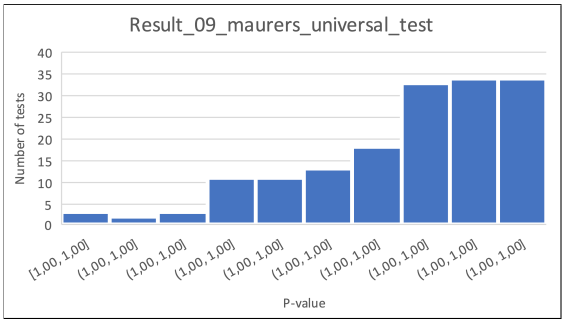

Result_06_dft_test
Number of tests
6000
5000
4000
3000
2000
1000
0
[0,00, 0,10]
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
(0,90, 1,00]
P-value

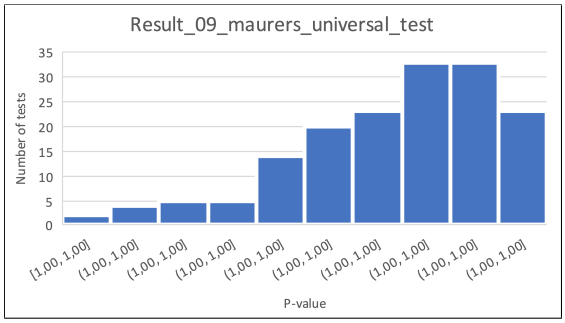

Result_07_non_overlapping_template_matching_test
Number of tests
4500
4000
3500
3000
2500
2000
1500
1000
500
0
[0,00, 0,10]
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
(0,90, 1,00]
P-value

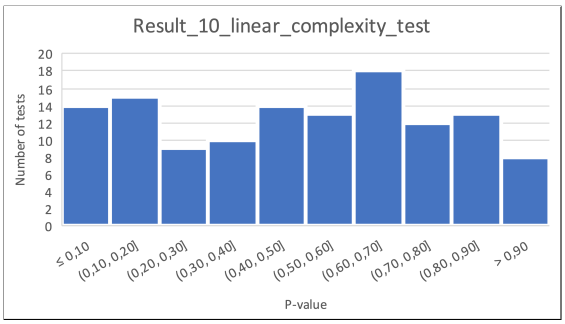

Result_07_non_overlapping_template_matching_test
Number of tests
4500
4000
3500
3000
2500
2000
1500
1000
500
0
[0,00, 0,10]
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
(0,90, 1,00]
P-value

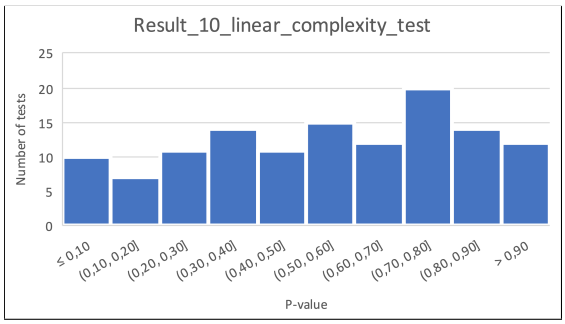

Result_08_overlapping_template_matching_test
Number of tests
16
14
12
10
8
6
4
2
0
≤ 0,10
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
> 0,90
P-value

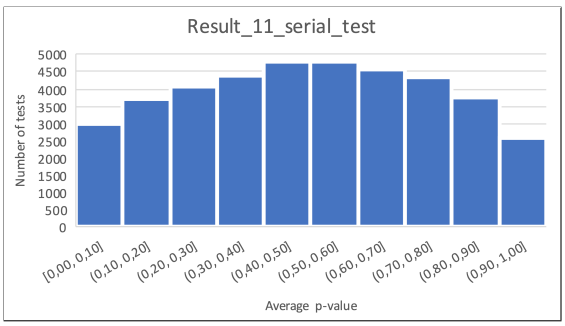

Result_08_overlapping_template_matching_test
Number of tests
12
10
8
6
4
2
0
[0,00, 0,10]
(0,10, 0,20]
(0,20, 0,30]
(0,30, 0,40]
(0,40, 0,50]
(0,50, 0,60]
(0,60, 0,70]
(0,70, 0,80]
(0,80, 0,90]
(0,90, 1,00]
P-value


Result_09_maurers_universal_test
Number of tests
P-value

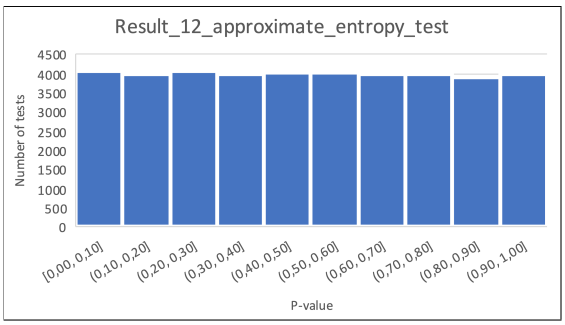

Result_09_maurers_universal_test
Number of tests
P-value

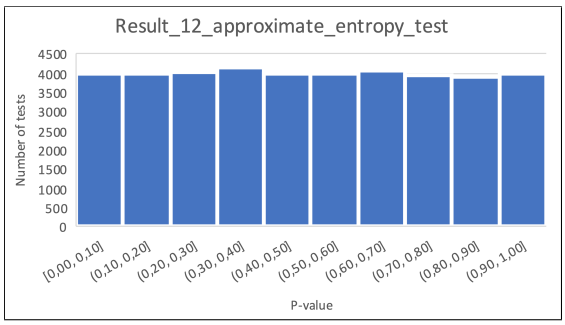

Result_10_linear_complexity_test
Number of tests
P-value

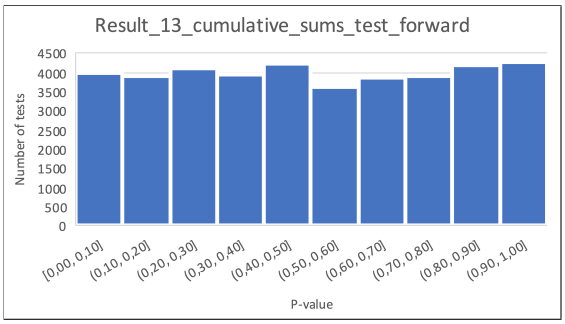

Result_10_linear_complexity_test
Number of tests
P-value

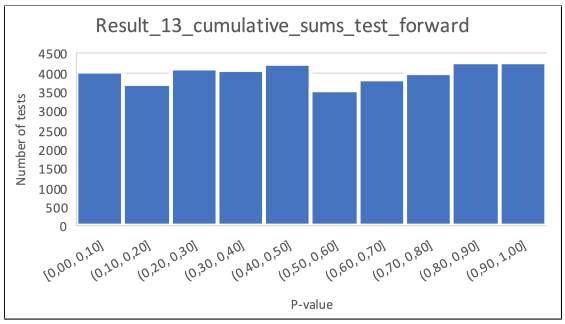

Result_11_serial_test
Number of tests
Average p-value

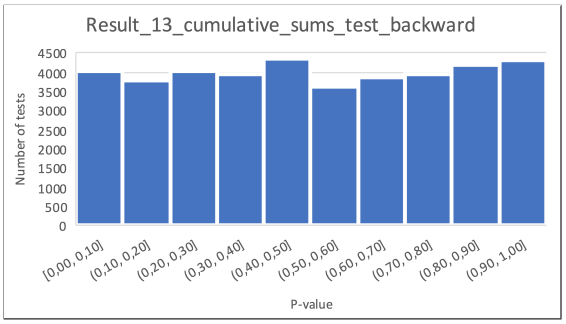

Result_11_serial_test
Number of tests
Averagep-value

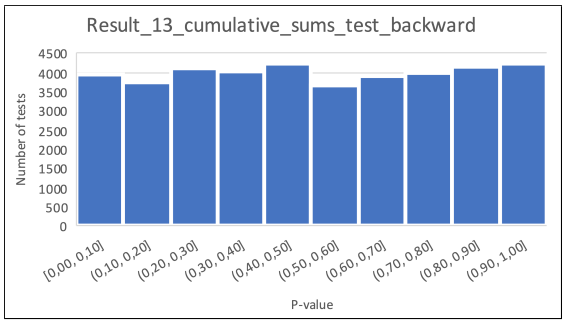

Result_12_approximate_entropy_test
Number of tests
P-value

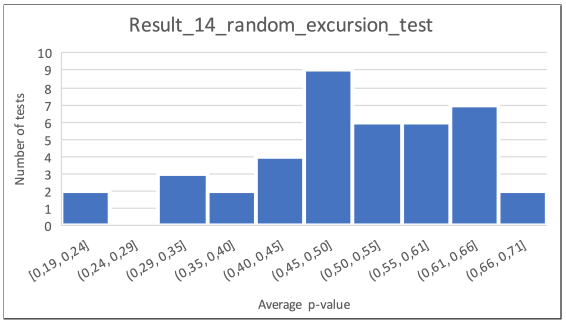

Result_12_approximate_entropy_test
Number of tests
P-value

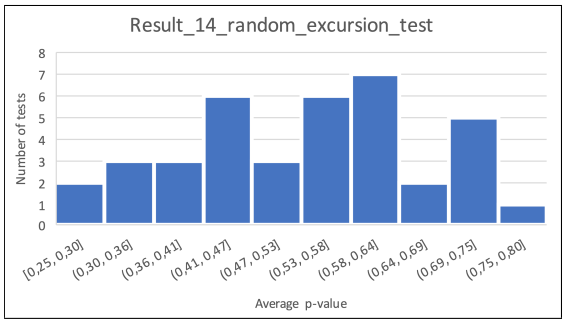

Result_13_cumulative_sums_test_forward
Number of tests
P-value

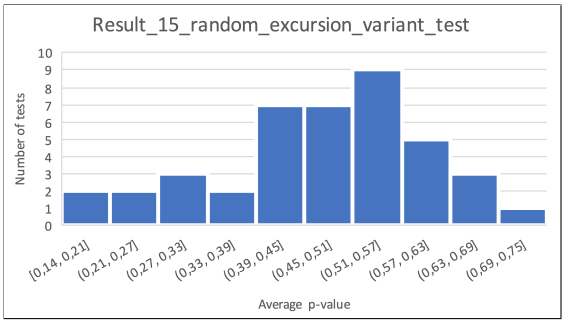

Result_13_cumulative_sums_test_forward
Number of tests
P-value

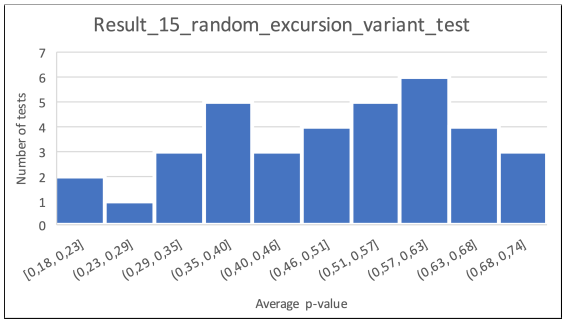

Result_13_cumulative_sums_test_backward
Number of tests
4500
4000
3500
3000
2500
2000
1500
1000
500
0
[0,00, 0,10]
[0,10, 0,20]
[0,20, 0,30]
[0,30, 0,40]
[0,40, 0,50]
[0,50, 0,60]
[0,60, 0,70]
[0,70, 0,80]
[0,80, 0,90]
[0,90, 1,00]
P-value


Result_13_cumulative_sums_test_backward
Number of tests
4500
4000
3500
3000
2500
2000
1500
1000
500
0
[0,00, 0,10]
[0,10, 0,20]
[0,20, 0,30]
[0,30, 0,40]
[0,40, 0,50]
[0,50, 0,60]
[0,60, 0,70]
[0,70, 0,80]
[0,80, 0,90]
[0,90, 1,00]
P-value


Result_14_random_excursion_test
Number of tests
10
9
8
7
6
5
4
3
2
1
0
[0,19, 0,24]
[0,24, 0,29]
[0,29, 0,35]
[0,35, 0,40]
[0,40, 0,45]
[0,45, 0,50]
[0,50, 0,55]
[0,55, 0,61]
[0,61, 0,66]
[0,66, 0,71]
Average p-value


Result_14_random_excursion_test
Number of tests
8
7
6
5
4
3
2
1
0
[0,25, 0,30]
[0,30, 0,36]
[0,36, 0,41]
[0,41, 0,47]
[0,47, 0,53]
[0,53, 0,58]
[0,58, 0,64]
[0,64, 0,69]
[0,69, 0,75]
[0,75, 0,80]
Average p-value


Result_15_random_excursion_variant_test
Number of tests
10
9
8
7
6
5
4
3
2
1
0
[0,14, 0,21]
[0,21, 0,27]
[0,27, 0,33]
[0,33, 0,39]
[0,39, 0,45]
[0,45, 0,51]
[0,51, 0,57]
[0,57, 0,63]
[0,63, 0,69]
[0,69, 0,75]
Average p-value


Result_15_random_excursion_variant_test
Number of tests
7
6
5
4
3
2
1
0
[0,18, 0,23]
[0,23, 0,29]
[0,29, 0,35]
[0,35, 0,40]
[0,40, 0,46]
[0,46, 0,51]
[0,51, 0,57]
[0,57, 0,63]
[0,63, 0,68]
[0,68, 0,74]
Average p-value

C Additional traffic table

Tab. C.1: Difference between original and new packet size (NORMAL mode)

Message name	Original size[bytes]	New size[bytes]	Additional traffic [%]
ASDB_VEHICLE	50	90	+80
ALTITUDE	44	84	+90
ATTITUDE	40	80	+100
ATTITUDE_TARGET	49	89	+81
BATTERY_STATUS	53	93	+75
CAMERA_IMAGE_CAPTURE	30	70	+133
COLLISION	31	71	+129
DEBUG	21	61	+190
DEBUG_FLOAT_ARRAY	32	72	+125
DEBUG_VECT	42	82	+95
DISTANCE_SENSOR	26	66	+153
ESTIMATOR_STATUS	54	94	+74
EXTENDED_SYS_STATE	14	54	+285
GLOBAL_POSITION_INT	40	80	+100
GPS2_RAW	47	87	+85
GPS_RAW_INT	42	82	+95
HOME_POSITION	64	104	+62
LOCAL_POSITION_NED	40	80	+100
NAMED_VALUE_FLOAT	30	70	+133
NAV_CONTROLLER_OUTPUT	38	78	+105
OBSTACLE_DISTANCE	170	210	+23
ORBIT_EXECUTION_STATUS	37	77	+108
PING	26	66	+153
POSITION_TARGET_GLOBAL_INT	63	103	+63
POSITION_TARGET_LOCAL_NED	63	103	+63
RC_CHANNELS	54	94	+74
SERVO_OUTPUT_RAW_0	33	73	+121
SYS_STATUS	43	83	+93
UMT_GLOBAL_POSITION	82	122	+49
VFR_HUD	32	72	+125
WIND_COV	52	92	+76
CERTIFICATE	182	/	/

D Current measurements on Pixhawk autopilot

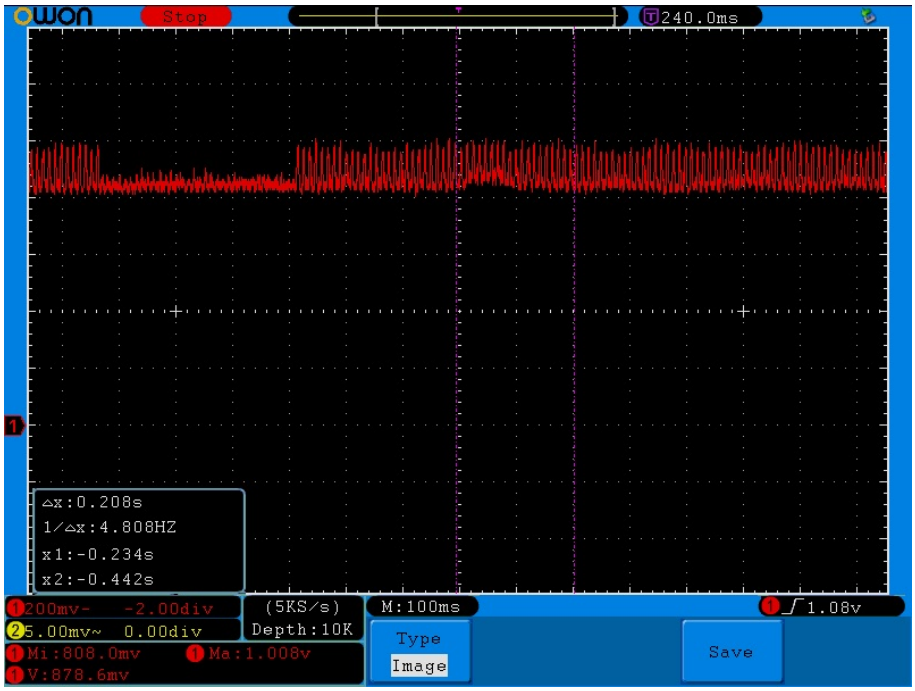


Fig. D.1: Voltage measured on resistor R1 (5.1 Ω) with new implementation

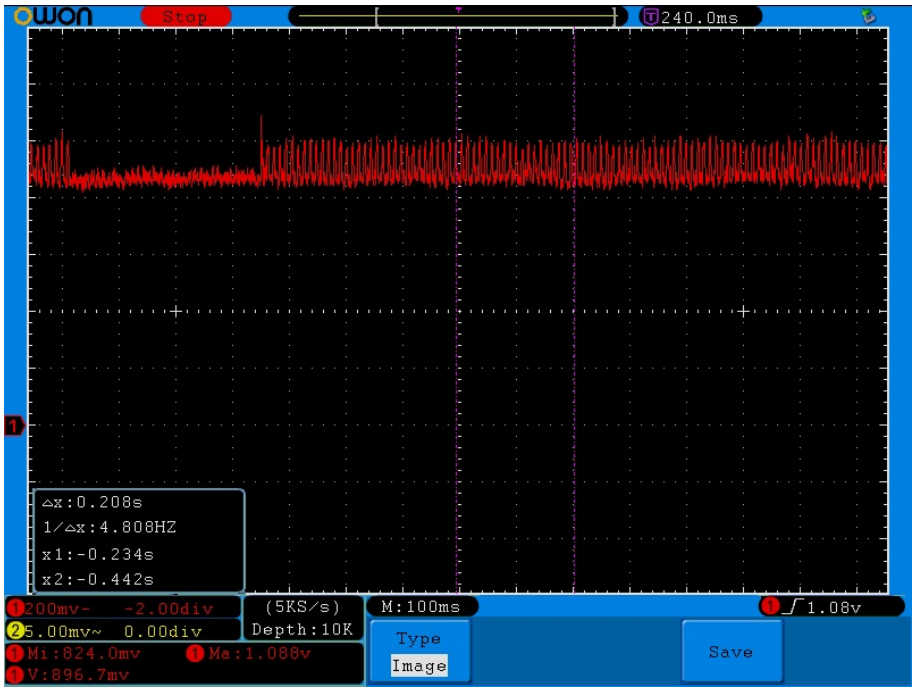


Fig. D.2: Voltage measured on resistor R1 (5.1 Ω) without new implementation

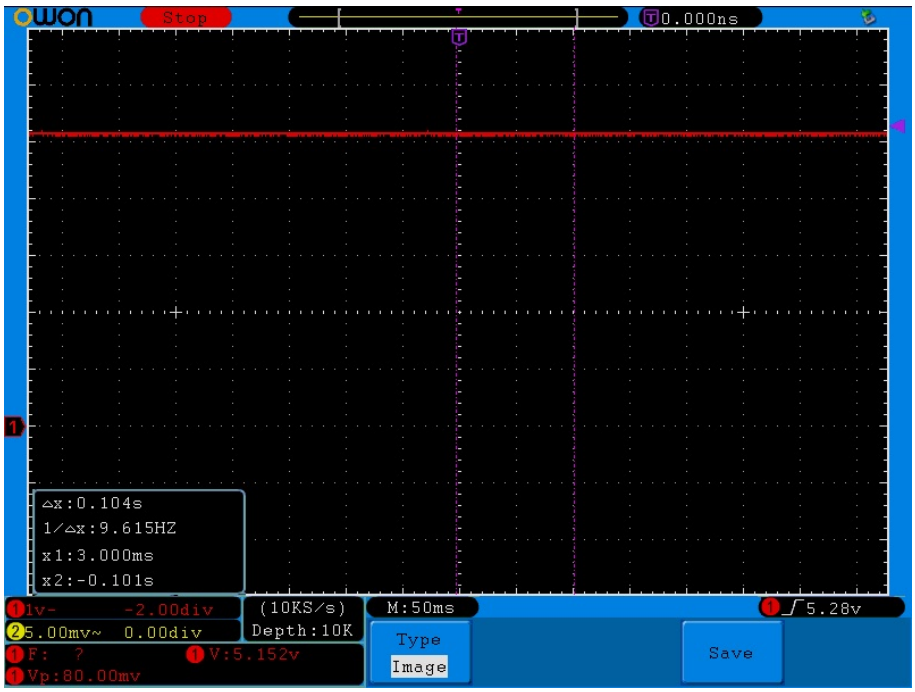


Fig. D.3: Voltage measured on power source.